(19)日本国特許庁(JP)　(12)公表特許公報(A)　(11)特許出願公表番号
特表2004-535611
(P2004-535611A)
(43)公表日　平成16年11月25日(2004.11.25)

(51)Int.Cl.[7]	F I			テーマコード（参考）
G06F 13/00	G06F	13/00	351Z	5B085
G06F 1/00	G06F	1/00	370E	5B089
G06F 15/00	G06F	15/00	330A	

審査請求 未請求　予備審査請求 有　（全 105 頁）

(21)出願番号　特願2002-562061 (P2002-562061)
(86)(22)出願日　平成14年1月29日 (2002.1.29)
(85)翻訳文提出日　平成15年7月28日 (2003.7.28)
(86)国際出願番号　PCT/GB2002/000385
(87)国際公開番号　WO2002/061552
(87)国際公開日　平成14年8月8日 (2002.8.8)
(31)優先権主張番号　0102516.2
(32)優先日　平成13年1月31日 (2001.1.31)
(33)優先権主張国　英国 (GB)
(81)指定国　EP (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), JP, US

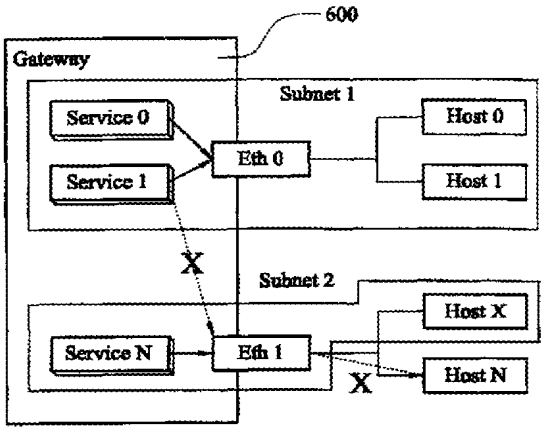
（特許庁注：以下のものは登録商標）
ＵＮＩＸ
Ｌｉｎｕｘ
ＪＡＶＡ

(71)出願人　398038580
ヒューレット・パッカード・カンパニー
ＨＥＷＬＥＴＴ－ＰＡＣＫＡＲＤ　ＣＯＭＰＡＮＹ
アメリカ合衆国カリフォルニア州パロアルト　ハノーバー・ストリート　３０００
(74)代理人　100081721
弁理士　岡田　次生
(74)代理人　100105393
弁理士　伏見　直哉
(74)代理人　100111969
弁理士　平野　ゆかり

最終頁に続く

(54)【発明の名称】高信頼性ゲートウェイシステム

(57)【要約】
アプリケーションの不正侵入によりもたらされる影響に対処する手段として、強制アクセス制御を組み込んだカーネル１００を備えたオペレーティングシステム。本オペレーティングシステムは、「コンテインメント」として知られる技法を用いて、セキュリティ侵害の発生時の損害範囲を少なくとも限度内に留める。好ましい実施形態では、本オペレーティングシステムによりサポートされる各アプリケーションには、論理的に保護されたコンピュータ処理環境すなわち「コンパートメントマルチサービス」をそれぞれ示すタグまたはラベルが割り当てられ、同じタグまたはラベルを有するアプリケーションは同じコンパートメントに属する。デフォルトにより、互いに通信することができるのは、同じコンパートメント中で実行されているアプリケーションのみである。アクセス制御規則により、非常に狭く、密に制御される通信経路がコンパートメント間に定義される。



【特許請求の範囲】
【請求項１】
内部および外部のネットワークの両方に接続されたデュアルインタフェースを備え、複数のサービス実行中プロセスをホストするゲートウェイシステムであって、論理的に保護されたコンピュータ処理コンパートメントを示すタグまたはラベルを前記実行中プロセスの少なくともいくつかに提供し、同じ前記タグまたはラベルを有する前記プロセスは、同じコンパートメントに属し、該システムは、前記コンパートメントと他のネットワークとの間に特定の通信経路をさらに定義し、通信経路またはインタフェース接続が間に定義されている場合にのみ、コンパートメントとホストまたは前記ネットワークとの間の通信を許可するゲートウェイシステム。
【請求項２】
前記ネットワークはローカルネットワークである請求項１記載のゲートウェイシステム。
【請求項３】
前記ネットワークはリモートネットワークである請求項１または２記載のゲートウェイシステム。
【請求項４】
前記通信経路はインタフェース接続を含む請求項１記載のゲートウェイシステム。
【請求項５】
どのクラスのプロセスがどのネットワークまたはホストにアクセス許可されているかを指定する規則を照会して、通信経路またはインタフェース接続が間に定義されているか否かを判定するようにするアクセス制御チェックを含む請求項１記載のゲートウェイシステム。
【請求項６】
前記アクセス制御チェックは、前記ゲートウェイシステムのオペレーティングシステムにおいて行われる請求項５記載のゲートウェイシステム。
【請求項７】
前記アクセス制御チェックは、前記ゲートウェイシステムの前記オペレーティングシステムのカーネルにおいて行われる請求項６記載のゲートウェイシステム。
【請求項８】
どのコンパートメントまたはプロセスが属するかを概念的に示すタグまたはラベルを、各実行中プロセスに添付する手段が設けられたカーネルを備える請求項５記載のゲートウェイシステム。
【請求項９】
コンパートメントと前記ホストまたはネットワークとの間で許可された前記通信経路を定義する規則を指定するシステム管理者を含む請求項８記載のゲートウェイシステム。
【請求項１０】
コンパートメントと前記ホストまたはネットワークとの間の許可された、前記通信経路を定義する、別個のコンパートメント単位のルーティング(rooting)テーブルが提供される請求項８記載のゲートウェイシステム。
【請求項１１】
前記実行中プロセスはスレッドである請求項８記載のゲートウェイシステム。
【発明の詳細な説明】
【技術分野】
【０００１】
本発明は、高信頼性オペレーティングシステムに関し、特に、アプリケーションの不正侵入および不正侵入されたアプリケーションの不正利用攻撃に対して高い保護性を有するオペレーティングシステムに関する。
【０００２】
近年、インターネットを介して電子的に提供されるサービスの数はますます増大している。かかるサービス、特に、成功している、ひいては収益性の高いサービスは潜在的な攻撃

者の標的となり、電子サービスを提供するアプリケーションが不正侵入され、その結果として多数のインターネットセキュリティ侵害が発生していることがわかっている。

【背景技術】

【０００３】

電子サービスを提供するアプリケーションは、一般に複雑であり、１つまたは複数のバグを内包することの多い多数のコードラインを含むことから、攻撃をより受けやすくなっている。電子サービスは、インターネット上で提供される場合には、サービスの脆弱性を探ることができる多数の潜在的な攻撃者に曝されており、かかるバグによりセキュリティの侵害があったことがわかっている。

【０００４】

アプリケーションが一度不正侵入される（たとえば、バッファオーバーフロー攻撃により）と、攻撃者はそのアプリケーションを異なるいくつかの方法で不正利用攻撃して、システムのセキュリティを侵害し得る。

【０００５】

単一のマシンを使用してマルチサービス（たとえば、ＩＳＰ、ＡＳＰ、ｘＳＰサービス提供）を同時にホストする場合がますます多くなっているため、アプリケーション不正侵入攻撃から保護されたホストプラットフォームのセキュリティのみならず、攻撃を受けた場合に、攻撃を受けたアプリケーションから他のアプリケーションを適宜保護することもますます重要になってきている。

【０００６】

オペレーティングシステムレベルにてアプリケーションの不正侵入から保護する最も有効な方法の１つは、カーネル実施制御であり、これは、カーネルで実施される制御には、いずれのアプリケーションまたはユーザによってもユーザ空間から乗っ取りまたは破壊を行うことができないためである。既知の諸システムでは、当該制御は、個々のアプリケーションコードの品質に関わりなくすべてのアプリケーションに適用される。

【０００７】

アプリケーションの不正侵入およびその影響から適宜保護するためには、システムレベルにおいて２つの基本的な要件がある。第１に、アプリケーションは、可能な限り最大の程度まで攻撃から保護されなければならず、公開されるアプリケーションへのインタフェースは可能な限り狭くなければならず、またかかるインタフェースへのアクセスは十分に制御されなければならない。第２に、不正侵入されたアプリケーションがシステムに与え得る損害の大きさは、可能な限り最大まで制限されなければならない。

【０００８】

既知のシステムでは、上記２つの要件は「コンテインメント（containment、封じ込め）」という抽象的プロパティによって満たされる。アプリケーションは、不正侵入されていた場合であっても、アクセス可能な資源および可能なアクセス種別が厳密に制御されている場合には封じ込められる。コンテインメントは、外部からの攻撃および干渉からもアプリケーションを保護する。このため、コンテインメントプロパティは、攻撃者の潜在的な不正利用攻撃行動の多くを少なくとも軽減する潜在性を有する。

【０００９】

アプリケーションの不正侵入に続く最も一般的な攻撃は、以下の４つのタイプの１つに概ね分類することができる（しかし、特定の攻撃の結果はこれらのうちのいずれかまたはすべての組み合わせであり得る）。

【００１０】

１．保護システム資源への直接アクセス権限を得る特権の悪用

アプリケーションが特別な特権で実行されている（たとえば、アプリケーションが標準Ｕｎｉｘオペレーティングシステム上でｒｏｏｔとして実行されている）場合、攻撃者はその特権を、意図された方法以外の方法で使用しようと企てることがある。たとえば、攻撃者はその特権を用いて、保護動作資源へのアクセスを得たり、または同じマシンで実行されている他のアプリケーションに干渉することができる。

【００１１】
２．アプリケーション実施アクセス制御の破壊
このタイプの攻撃は、正当な資源（すなわち、アプリケーションによって公開されるものと意図される資源）へのアクセスを不正に得る。たとえば、コンテンツを提供する前にコンテンツにアクセス制御を実施するウェブサーバは、このタイプの攻撃を受けやすいアプリケーションの１つである。ウェブサーバのコンテンツへのアクセスは非制御直接アクセスであるため、ウェブサーバの制御権を得る攻撃者にとっても同様である。
【００１２】
３．誤ったセキュリティ意思決定情報の付与(supply)
通常、このタイプの攻撃は間接的な攻撃であり、不正侵入されたアプリケーションは通常、メインサービスとは対照的なサポートサービス（認証サービス等）である。この場合、不正侵入したセキュリティサービスを使用して、誤っているか、または偽作の情報を与え、それによって攻撃者がメインサービスへのアクセスを得ることが可能になる。したがって、これは、攻撃者がアプリケーションにより正当に公開されている資源に不正アクセスすることができるもう一つの方法である。
【００１３】
非保護システム資源の不正使用
攻撃者が、保護されていないが、通常はアプリケーションによって公開されない、マシンのローカル資源にアクセスする。通常、次にかかるローカル資源を使用して、さらなる攻撃を開始する。たとえば、攻撃者はホストシステムへのシェルアクセスを取得することができ、次いでそこから、マシン上またはネットワークにわたる他のアプリケーションに対して段階的な攻撃を開始することができる。
【００１４】
コンテインメントを使用する場合、保護システム資源への直接アクセスを得る特権の悪用の影響は、コンテインメントを使用しない場合よりもはるかに少ない。これは、攻撃者がアプリケーション特権を利用する場合であっても、アクセスされる恐れのある資源は、アプリケーションのコンテナ内で利用可能になっていたものに制限することができるためである。同様に、非保護資源の場合でも、コンテインメントを使用すると、アプリケーションからネットワークへのアクセスを阻止、または少なくとも非常に密に制御することができる。誤ったセキュリティ意思決定情報の付与に関しては、コンテインメントでは、サポートサービスへのアクセスを確実に正当なクライアント、すなわちアプリケーションサービスのみからにし、それによって、アプリケーションが攻撃に曝されることを制限することにより、引き起こされる潜在的な損害が少なくなる。
【００１５】
第２のタイプの攻撃、すなわちアプリケーション実施アクセス制御の破壊の軽減または防止は通常、アプリケーションの設計において、または少なくとも構成レベルにおいてなされる。しかし、コンテインメントを使用すれば、信頼性の低い大きなアプリケーション（ウェブサーバ等）から保護資源へのアクセスが、より小さくより信頼性の高いアプリケーションを経由しなければならないように構成することが可能である。
【００１６】
このため、コンテインメントをオペレーティングシステムに使用すると、アプリケーションのセキュリティが効果的に向上し、アプリケーションが不正侵入された場合に攻撃者によって引き起こされる恐れのあるあらゆる損害が制限される。図面の図１を参照して、コンテインメントプロパティを有するオペレーティングシステム上でマルチサービスホストとして機能するための例示的なアーキテクチャを示す。図示の例では、コンテインメントは、アプリケーションが相互に、またクリティカルなシステム資源から分離された状態に保たれるよう保証するために使用されている。アプリケーションは、別のアプリケーションの処理に干渉すること、またはその（おそらく機密の）データにアクセスすることができない。コンテインメントが使用されると、特定のアプリケーションが機能させる必要があるインタフェース（入力および出力）のみがオペレーティングシステムによって公開さ

れるよう保証されるため、特定のアプリケーションに対する攻撃の範囲が制限されるとともに、アプリケーションが不正侵入されたときに生じ得る損害量が制限される。したがって、コンテインメントはホストプラットフォーム全体の保全性の維持に役立つ。

【００１７】

オペレーティングシステム内のカーネル実施コンテインメント機構は、数年前から、通常、機密（軍事）情報を受け渡し・処理するために設計されたオペレーティングシステムに利用可能であった。多くは、かかるオペレーティングシステムは「高信頼性オペレーティングシステム」と呼ばれる。

【００１８】

コンテインメントプロパティは通常、強制アクセス制御（ＭＡＣ）と特権との組み合わせを通して実現される。ＭＡＣ保護方式は、特定のアクセス制御ポリシーをファイル、プロセス、およびネットワーク接続等のシステム資源に対して実施する。このポリシーはカーネルによって実施され、ユーザまたは不正侵入されたアプリケーションによって乗っ取ることができない。

【００１９】

高信頼性オペレーティングシステムは、魅力的なコンテインメントプロパティを提供するにも関わらず、主に２つの理由により、機密情報処理システム以外ではあまり広く使用されていない。第１に、従来では、高信頼性オペレーティングシステム機構を従来のオペレーティングシステムに追加しようとすると、標準アプリケーションまたは管理ツールがサポートされなくなり、標準の方法で使用または管理することができなくなり得るという意味において、通常、土台をなすオペレーティングシステムの個性が失われることになる。このため、高信頼性オペレーティングシステムはそれぞれの標準対応品よりもはるかに複雑である。第２に、従来の高信頼性オペレーティングシステムは、通常、分離により近い、すなわち強すぎるコンテインメントの形態を動作させるため、大がかりで高価なことが多い統合努力を伴うことなく、（既存の）アプリケーションを有用かつ効果的にセキュア化する機能に関して範囲が限られていることがわかっていた。

【００２０】

本発明者らによる第１の同時係属中の国際出願では、上記問題の克服を目的とし、アプリケーションを変更することなく多数の既存のアプリケーションを効果的にセキュア化するために有効に使用することができるコンテインメントプロパティを有する高信頼性オペレーティングシステムを提供する装置が定義されている。

【００２１】

本発明者らによる第１の同時係属中の国際出願の発明の第１の態様は、複数のアプリケーションをサポートするオペレーティングシステムであって、上記アプリケーションの少なくともいくつかにはラベルまたはタグが提供され、ラベルまたはタグはそれぞれ上記システムの論理的に保護されたコンピュータ処理環境すなわち「コンパートメント」を示し、同じラベルまたはタグを有するアプリケーションはそれぞれ同じコンパートメントに属するオペレーティングシステムを提供し、オペレーティングシステムは、上記コンパートメント間に１つまたは複数の通信経路を定義する手段と通信経路が間に定義されないコンパートメント間の通信を阻止する手段とをさらに備える。

【００２２】

本発明者らによる第１の同時係属中の国際出願の発明の第２の態様は、複数のアプリケーションをサポートするオペレーティングシステムであって、複数のアクセス制御規則をさらに備えるオペレーティングシステムを提供する。複数のアクセス制御規則は、ユーザ空間から都合良く追加することができ、オペレーティングシステムのカーネルに設けられる手段によって実施され、選択されたアプリケーション（上記オペレーティングシステムに対してローカルであるかリモートであるかに関わりなく）間の通信インタフェースのみを定義する。

【００２３】

これは、本発明者らによる第１の同時係属中の国際出願の発明の第１および第２の態様に

おいて、コンテインメントプロパティが、プロセス、ファイル、およびネットワーク資源の強制保護によって提供され、主要な概念は、システムの半ば隔離された部分であるコンパートメントに基づく。システム上のサービスおよびアプリケーションは、別個のコンパートメント内で実行される。都合のよいことに、各コンパートメント内はホストファイルシステムの限られた部分集合であり、各コンパートメントを対象とする通信インタフェースは、明確に定義され、狭く、密に制御される。各コンパートメント内のアプリケーションのみがそのコンパートメント内の資源、すなわちそのコンパートメント内の限られたファイルシステムおよび他のアプリケーションに直接アクセスする。他の資源へのアクセスは、ローカルであれリモートであれ、十分に制御された通信インタフェースを介してのみ提供される。

【００２４】

単純な強制アクセス制御、およびアプリケーションもしくはプロセスのラベリングを都合良く使用して、コンパートメントの概念を実現する。好ましい実施形態では、各プロセス（またはスレッド）にラベルが与えられ、同じラベルを有するプロセスは同じコンパートメントに属する。システムは、好ましくは、強制セキュリティチェックを行って、あるコンパートメントからのプロセスが別のコンパートメントからのプロセスに干渉することができないよう保証する手段をさらに備える。ラベルは整合するかしないかのいずれかであるため、アクセス制御は非常に簡単に行うことができる。

【００２５】

本発明の好ましい実施形態では、ファイルシステム保護もまた強制である。従来の高信頼性オペレーティングシステムとは異なり、本発明者らによる第１の同時係属中の国際出願の発明の第１の態様の好ましい実施形態は、ファイルシステムへのアクセスの直接制御にラベルを使用しない。代わりに、本発明者らによる第１の同時係属中の国際出願の発明の第１および第２の態様のファイルシステムは、好ましくは、少なくとも部分的にセクションに分割される。各セクションは、メインファイルシステムの重ならない、限られた部分集合（すなわち、ｃｈｒｏｏｔ）であり、各コンパートメントに関連する。各コンパートメントで実行されているアプリケーションのみが、ファイルシステムの関連セクションにアクセスする。本発明者らによる第１の同時係属中の国際出願の発明の第１および／または第２の態様のオペレーティングシステムには、好ましくは、本発明者らによる第１の同時係属中の国際出願の発明の第４の態様を参照して以下に述べるように、ｃｈｒｏｏｔをエスケープすることができないように、プロセスがコンパートメント内からｒｏｏｔに移行することを阻止する手段が設けられる。本システムは、ｃｈｒｏｏｔ内の選択ファイルを変更不可にする手段も含むことができる。

【００２６】

コンパートメント間、およびネットワーク資源間の柔軟であるが制御される通信経路は、狭く密に制御される通信インタフェースを通して提供され、通信インタフェースは、セキュリティ管理者等により好ましくはコンパートメント単位でユーザ空間から定義・追加が可能な１つまたは複数の規則によって統制される。かかる通信規則により、コンパートメントおよび／またはネットワーク資源の間の通信を許可する高信頼性プロキシの必要性がなくなる。

【００２７】

本発明者らによる第１の同時係属中の国際出願の発明の第１および／または第２の態様によって提供されるコンテインメントプロパティは、カーネルレベルの実施手段、ユーザレベルの実施手段、またはこれら２つの組み合わせによって実現することができる。本発明者らによる第１の同時係属中の国際出願の発明の第１および／または第２の態様の好ましい実施形態では、あるコンパートメントと他のコンパートメントまたはホストとの間のアクセス許可を指定するために使用される規則は、オペレーティングシステムのカーネル内の手段によって実施されるため、ユーザ空間介入の必要性（既存のプロキシソリューションに必要なもの等）がなくなる。カーネル実施コンパートメントアクセス制御規則により、アプリケーションを変更する必要なく、本発明者らによる第１の同時係属中の国際出願

の発明の第１の態様のコンパートメント化されたオペレーティングシステムにおけるコンパートメント間の制御された柔軟な通信経路が可能になる。

【００２８】

規則の有益な形は以下である。

【００２９】

source->destination method m[attr][netdev n]

ただし、ｓｏｕｒｃｅ／ｄｅｓｔｉｎａｔｉｏｎは、以下のうちの１つである。

【００３０】

ＣＯＭＰＡＲＴＭＥＮＴ（名前の付いたコンパートメント）

ＨＯＳＴ（おそらく固定Ｉｐｖ４アドレス）

ＮＥＴＷＯＲＫ（おそらくＩｐｖ４サブネット）

ｍ：サポートされるカーネル機構、たとえば、ｔｃｐ（伝送制御プロトコル）、ｕｄｐ（ユーザデータグラムプロトコル）、ｍｓｇ（メッセージキュー）、ｓｈｍ（共有メモリ）等

ａｔｔｒ：メソッドｍをさらに修飾するプロパティ

ｎ：妥当な場合には、名前の付いたネットワークインタフェース、たとえばｅｔｈ０

規則の指定にワイルドカードを使用することも可能である。以下の規則例は、ＴＣＰを使用してポート８０のみですべてのホストがウェブサーバコンパートメントへアクセスすることを許可する。

【００３１】

HOST*->COMPARTMENT web METHOD tcp PORT80

以下の規則例はかなり似ているが、ウェブサーバコンパートメントへのアクセスを、システムの例示的な実施形態においてｅｔｈ０ネットワークインタフェースへのルートを有するホストに制限する。

【００３２】

HOST*->COMPARTMENT web METHOD tcp PORT80 NETDEV eth0

有益には、許可を受けたシステム管理者によって、オペレーティングシステムに規定されたアクセス制御規則の追加、削除、および／またはリスト化を行う手段が設けられることが好ましい。選択されたコンパートメント間および／または資源間で双方向通信を行うことができるようにするために、リバースＴＣＰ規則を追加する手段を設けてもよい。

【００３３】

規則は、有益なことに、カーネルレベルのデータベースに格納され、好ましくは、ユーザ空間から追加される。カーネルレベルのデータベースは、有益なことに、２つのハッシュテーブルで構成され、テーブルの一方は規則の発信元アドレス詳細に焦点をあて、他方は規則の宛先アドレス詳細に焦点をあてる。システムは、システムコール／ＩＳＲ（割り込みサービスルーチン）の処理を許可する前に、データベースをチェックして規則が適切な通信経路を定義しているか否かを判定するように構成される。カーネルレベルデータベースの好ましい構造により、セキュリティチェックを行う時に、システムは、要求される規則が発信元アドレス詳細または宛先アドレス詳細に整合すべきかどうかを知っており、よって適切なハッシュテーブルを選択することができ、規則参照のＯ（１）レートを可能にするため、カーネル実施コンパートメントアクセス制御規則を効率的に参照することができる。要求される通信経路を定義する必要な規則が見つからない場合には、システムコールは失敗することになる。

【００３４】

本発明者らによる第１の同時係属中の国際出願の発明の第３の態様は、複数のアプリケーションをサポートするオペレーティングシステムであって、上記アプリケーション間の許可された通信経路（すなわち、発信元および宛先）を定義する複数の規則が格納されたデ

ータベースを備えるオペレーティングシステムを提供する。上記規則は、少なくとも２つの符号化テーブル、すなわち規則の発信元詳細に焦点をあてた第１のテーブルおよび規則の宛先詳細に焦点をあてた第２のテーブルの形態で格納される。本システムは、システムコールに応答して、要求される通信経路を定義する規則の存在について上記テーブルの少なくとも一方をチェックし、上記要求された通信経路が定義されている場合にのみ、上記システムコールの続行を許可する手段をさらに備える。

【００３５】

上記符号化テーブルは、好ましくは、少なくとも１つのハッシュテーブルを含む。

【００３６】

多くの場合、ゲートウェイタイプのシステム（すなわち、内部および外部両方のネットワークに接続されたデュアルインタフェースを備えたホスト）では、以下のことが望ましい：ａ）利用可能なネットワークインタフェースの部分集合のみを使用するように実行中のサーバプロセスを制限すること、ｂ）アクセス可能なリモートホストおよびアクセス不可能なリモートホストを明示的に指定すること、およびｃ）かかる制限をプロセス／サービス単位で同じゲートウェイシステムに適用させること。

【００３７】

ゲートウェイシステムは、いくつかの内部サブネットワークに物理的に接続されることがあるため、サーバプロセスがリモートソースから不正侵入された場合に、別のネットワークインタフェースを介して、潜在的に脆弱性を有するバックエンドホストへのその後の攻撃開始に使用することができないように、システム管理者が、どのサーバプロセスがどのネットワークインタフェースにアクセス許可され得るかを分類することが重要である。

【００３８】

従来では、ＩＰアドレス単位および／またはＩＰポートレベル単位でホスト間のアクセスを制限するようにファイアウォールが使用されてきた。しかし、かかるファイアウォールは、主に、異なるサーバプロセスを区別することができないという理由により、複数のサービスをホストするゲートウェイシステムに十分なきめ細かさを提供しない。加えて、異なる制限セットを指定するには、別個のファイアウォール規則のセットを有する別個のゲートウェイシステムが求められる。

【発明の開示】

【発明が解決しようとする課題】

【００３９】

ここで、上記問題の克服を目的とした装置を考案した。

【課題を解決するための手段】

【００４０】

このように、本発明によると、内部および外部ネットワークの両方に接続されたデュアルインタフェースを備え、プロセスおよび／またはスレッドを実行している複数のサービスをホストするゲートウェイシステムが提供される。このシステムは、コンパートメントを示すタグまたはスレッドを上記実行中のプロセスおよび／またはスレッドの少なくともいくつかに提供する手段を備え、同じタグまたはラベルを有するプロセス／スレッドは同じコンパートメントに属する。このシステムは、上記コンパートメントとローカルおよび／またはリモートホストもしくはネットワークとの間に特定の通信経路および／または許可されたインタフェース接続を定義する手段と、通信経路またはインタフェース接続が間に定義されている場合にのみ、コンパートメントとホストまたはネットワークとの間の通信を許可する手段とをさらに備える。

【００４１】

かくして、本発明では、アクセス制御チェックが、好ましくはゲートウェイシステムのカーネル／オペレーティングシステムに対して行われる。かかるアクセス制御チェックでは、好ましくは、どのプロセスクラスがどのサブネット／ホストにアクセス許可されているかを指定する規則テーブルを照会する。制限は、サービス（またはプロセス／スレッド）レベル単位で指定することができる。これは、バックエンドネットワークのビューが単一

のゲートウェイホスト上で可変であることを意味する。したがって、たとえば、ゲートウェイが、２つの異なるバックエンドホストへのアクセスをそれぞれ要求する２種類のサービスをホストする場合、従来技術によるファイアウォールは、ゲートウェイホストがこれらバックエンドホストの両方にアクセス可能なことを指定する必要があるが、本発明では、許可された通信経路をより細かいレベルで、すなわちどのサービスがどのホストにアクセス許可されているかを指定することが可能である。これにより、サービスが当初アクセスすることが意図されていなかったホストにアクセスする危険性が大幅に低下するため、セキュリティはいくらか向上する。

【００４２】

本発明の好ましい実施形態では、アクセス制御チェックはゲートウェイシステムのカーネル／オペレーティングシステムにおいて実施されるため、ユーザ空間プロセスがこれを迂回することはできない。

【００４３】

かくして、本発明の第１の例示的な実施形態では、ゲートウェイシステムのカーネルに、プロセスがどのコンパートメントに属するかを概念的に示すタグまたはラベルを実行中の各プロセス／スレッドに添付する手段が設けられる。かかるタグは、子にフォークする親プロセスから受け継ぐことができる。したがって、スレーブウェブサーバプロセス群等、協働して作業負荷を分担する、フォークした子の群を含むサービスは、同じタグを保有し、同じ「コンパートメント」に配置される。システム管理者は、たとえば、以下の形態で規則を指定することができる。

【００４４】

Compartment X->Host Y[using Network Interface Z]または

Compartment X->Subnet Y[using Network Interface Z]

これらは、名前の付いたコンパートメントＸ中のプロセスによる、ホストあるいはサブネットＹへのアクセスを許可し、オプションとして、Ｚと名の付いたネットワークインタフェースのみを使用する場合に制限される。好ましい実施形態では、かかる規則は、ゲートウェイシステム上の安全な構成ファイルに格納され、システムスタートアップ時に、次いて開始されるサービスが実行することができるように、カーネル／オペレーティングシステムにロードされる。サービスが開始されると、それぞれのスタートアップシーケンスが、最初にどのコンパートメントに配置されるかを指定する。この実施形態では、好ましくはカーネルのプロトコルスタックにおいてセキュリティチェックをさらに行うことにより、パケットをコンパートメントＸから送信またはコンパートメントＸに引き渡す都度、規則を照会する。

【００４５】

本発明の第２の例示的な実施形態では、別個のルーティングテーブルがコンパートメント毎に提供される。上に述べた第１の実施形態と同様に、各プロセスは、親から受け継いだタグまたはラベルを保有する。特定の名前の付いたプロセスは、システム管理者によって構成された指定タグで始まる。規則を指定する代わりに、第１の例示的な実施形態を参照して上述したように、所望のルーチンテーブルエントリを挿入することにより、各コンパートメントのルーティングテーブルを構成する構成ファイルのセットが提供される（各コンパートメントに１つ）。ゲートウェイシステムはいくつかの名前の付いていないコンパートメントを含むことができるため、各コンパートメントのルーティングテーブルはデフォルトでは空である（すなわち、エントリがない）ことが好ましい。

【００４６】

整合するルートがないことは、到達を試みられているリモートホストが到達不可能であると報告されることを意味すると解釈されるため、明示的な規則の代わりにルーティングテーブルを使用することができる。整合するルートは、そのリモートホストにアクセスしようという試みが許可されることを意味する。上に述べた第１の例示的な実施形態における規則と同様に、ルーティングエントリは、ホスト単位（ＩＰアドレス）またはサブネット

単位で指定することができる。第１の例示的な実施形態と同じ機能を実現するために必要なのは、かかるルーティングエントリをコンパートメント単位で指定することだけである。

【００４７】

上に述べたように、実行中のサーバプロセス／デーモンへの攻撃（たとえば、バッファオーバーフロー、スタック破壊）は、遠隔地にいる攻撃者が、サーバプロセスをホストしているシステムでｒｏｏｔ／管理者に相当するアクセスを不法に得る状況に繋がる恐れがある。かかるシステムで管理者アクセスを得ると、攻撃者は、不正侵入されたシステムに存在することがある機密の構成／パスワードファイル、非公開データベース、秘密鍵等の読み出しなど、他のセキュリティ侵害を自由に開始することができる。

【００４８】

かかる攻撃は、以下の場合に可能であり得る。

【００４９】

ａ）サーバプロセスが管理者として実行され、ソフトウェアバグにより実行時に内部に侵入される。

【００５０】

ｂ）サーバプロセスが、最初は管理者として開始されるが、ある特権動作を行うのに先立って管理者特権を再び取得する選択的機能を備えて、その動作時間の大半は管理者特権を落とすようにプログラムされた。かかる場合、サーバプロセスは（ある特定の目的のために）ｒｏｏｔに移行して戻る機能を保持するが、攻撃者は、プロセスの制御権をいったん取得すると、意図された当初の目的以外でｒｏｏｔへの移行を行うことができる。

【００５１】

ｃ）サーバプロセスが、最初は非特権ユーザとして開始されるが、まずオリジナルのサーバプロセスを破壊し、次いでそれを、上述した方法で脆弱性を有する可能性がある外部ｓｅｔｕｉｄ－ｒｏｏｔプログラムを破壊する手段として使用することによって管理者アクセスを得る。

【００５２】

従来技術によれば、こういった問題に対する１つの直接的な方策は、最初に攻撃の発生を許した特定のバッファオーバーフローバグを塞ぐ／修復することである。この方策の明らかな欠点はもちろん、純粋に受動的であり、バッファオーバーフローバグが今後さらに見つかり、不正利用攻撃される事態が防止されないことである。従来技術により提案された別の方策は、オペレーティングシステム、たとえばＵｎｉｘに存在する機能について、決して戻さないという意図をもって、ｒｏｏｔに相当するアクセスをすべて落とすというものである。これにより、実行中のプロセスが予期せずにｒｏｏｔに復帰する事態が回避されるが、プログラムが、たとえば、不注意に周辺に転がっており、ある不正入力が与えられると破られやすい外部ｓｅｔｕｉｄ－ｒｏｏｔプログラムを動作させるという事態を回避しない。これが万が一行われた場合、不正侵入された、非特権ユーザとして実行中のプロセスは、ｓｅｔｕｉｄ－ｒｏｏｔプログラムを実行し、攻撃者の制御下にさせる入力を与えるという事態を防止しない。

【００５３】

上記問題を克服することを目的とした装置を考案した。このように、本発明者らによる第１の同時係属中の国際出願の発明の第４の態様は、複数のアプリケーションをサポートするオペレーティングシステムであって、要求に応答して、アプリケーションがｒｏｏｔに移行することが許可されているか否かを示すタグまたはラベルを上記アプリケーションの少なくともいくつかに提供する手段と、かかる要求を識別し、そのタグまたはラベルから、アプリケーションのｒｏｏｔへの移行が許可されているか否かを判定し、判定に応じて上記移行を許可または拒絶する手段とを備えるオペレーティングシステムを提供する。

【００５４】

好ましい実施形態では、上記タグまたはラベルの少なくとも１つは、そのタグまたはラベルが添付されている、または関連付けられたアプリケーションが「封印」されたもの、ひ

いては変更不可であることを示す。

【００５５】

このように、本発明者らによる第１の同時係属中の国際出願の発明の第４の態様は、選択されたサーバプロセスを、管理者相当状態への状態移行に関して「封印」することにより、かかるサーバプロセスの管理者相当状態への移行を止める方法を導入する。こういったプロセスがかかる移行を行おうとすると常に、かかる目的専用のシステムルーチンを呼び出すか、「ｓｅｔｕｉｄ－ｒｏｏｔ」とマークされた外部プログラム（すなわち、呼び出した人が誰であれ、管理者として実行する機能を有するものとして、システム管理者により予めタグを付けられているプログラム）を実行するか、あるいはあらゆる他の手段により、オペレーティングシステムは、このようにマークされたプログラムを実行するシステムコールまたは試行を許可しない。

【００５６】

本発明者らによる第１の同時係属中の国際出願の発明の第４の態様によるオペレーティングシステムによってもたらされる利点としては、ｒｏｏｔに相当するアクセスに対する制限が無条件であり、実行されるサーバプロセスに、不正利用攻撃されるまだ発見されていないソフトウェアバグがいくつあるかに関わりなく、効力を失わない状態を保つことが挙げられる。不正利用攻撃される恐れのある新しいバグが発見された場合、制限は、新しいバグの性質に関わりなく、それまで他のバグの場合と同じように課された状態のままである。明らかに、これは、バグが発見されたときにそのバグを修復する必要がある場合には起こりえない。さらに、本発明者らによる第１の同時係属中の国際出願の発明の第４の態様の装置は、攻撃者が、オリジナルプロセスの代わりにｒｏｏｔとして実行する機能を有する外部プログラムの破壊を企てる外部ｓｅｔｕｉｄ－ｒｏｏｔ問題を解消している。本発明者らによる第１の同時係属中の国際出願の発明の第４の態様の装置では、オペレーティングシステムにおいてかかるあらゆる企てが追跡され、装置は、マークされたプロセスを用いてかかるｓｅｔｕｉｄ－ｒｏｏｔプログラムを実行する試みを拒絶するように構成することができる。加えて、保護プロセスの元のソースコードを変更する必要はなく、ｒｏｏｔに戻らないことを保証して、任意のバイナリを実行することができる。

【００５７】

高信頼性オペレーティングシステムは通常、入力ネットワークパケットに割り当てる必要のある機密ラベルの決定に役立つように、個々のネットワークアダプタのラベル付けを行う。ファイアウォール等他のソフトウェアシステムが、どのインタフェースを潜在的な「敵」または非敵とマークすべきかを決定するために、インタフェースラベル付け（またはカラーリングと呼ばれることもある）を実行することがある。これは、内部では高信頼性／安全であり、外部インターネットリンクに関しては低信頼性／非安全な企業ネットワークのビューに対応する（図面の図１５を参照）。

【００５８】

コンピュータシステムの動作中静的なままであるネットワークアダプタ（ＮＩＣ）の場合、ラベル付けはシステムスタートアップ中に行うことができる。しかし、ＰＰＰリンクまたはあらゆる他のネットワークデバイス抽象化（たとえば、ＶＬＡＮ、ＶＰＮ）を処理する「ソフト」アダプタ等、システム上で動的にアクティブ化することのできる類のＮＩＣがある。かかる動的アダプタの例としては、以下が挙げられる。

【００５９】

＊ＰＰＰリンク、たとえばＩＳＰへのモデム接続。通常、ＩＳＰへのＰＰＰ接続を表すソフトアダプタが作成される。

【００６０】

＊仮想ＬＡＮ（ＶＬＡＮ）－サーバは、ＶＬＡＮを使用して、私設仮想回線で動作するソフトウェアサービスをホストすることができる。かかるＶＬＡＮは動的に（たとえば、要求があり次第）セットアップすることができるため、高信頼性オペレーティングシステムまたはそれに由来するものを使用する場合、かかるサービスをホストするサーバがこういったインタフェースを正しくラベル付けることができるはずである。

【００６１】
図面の図１５に示す構成が概ね静的な性質であることは、新しいアダプタを処理する必要が殆どないことを意味する。システム管理者は、新しいアダプタをデュアルホームホスト７００に追加したい場合には、通常システムを停止し、アダプタを物理的に追加して、新しいアダプタを適宜認識するようにシステムを構成する。しかし、このプロセスは、インタフェースラベル付けを要求するシステムが上に述べた種類の動的インタフェースを有する場合には適さない。
【００６２】
ラベルがアダプタに付けられない場合、アダプタ上の入力パケットに正しいラベルが割り当てられず、問題となっているシステムのセキュリティに違反することがある。さらに、出力パケット（おそらく、ラベルが正しく割り当てられている）は、パケットを送信すべきアダプタと正しく整合しえないため、問題となっているシステムのセキュリティに違反する。
【００６３】
上記問題の克服を目的とした装置を考案した。このように、本発明者らによる第２の同時係属中の国際出願は、新たにインストールされたアダプタを実質的にアクティブ化するときに、上記アダプタの属性に依存するラベルをそのアダプタに動的に割り当てる手段と、上記アダプタが非アクティブ化されるときに上記ラベルを除去する手段とを備えるオペレーティングシステムを提供する。
【００６４】
このように、オペレーティングシステムに新たにインストールされたアダプタが初めてアクティブ化されると、入力パケットの受け取りに先立ってラベルが確実に割り当てられ、それによってラベルの付いていないパケットが作成されないこと、またネットワークプロトコルスタックに渡されないことが保証される。動的アダプタは、本発明者らによる第２の同時係属中の国際出願の発明のオペレーティングシステムにおいて作成されるため、かかるラベルシステムの新しい機能分野が、たとえば、ルータ、モバイル機器として開かれる。さらに、アダプタに割り当てられるラベルは、新たにアクティブ化されたアダプタの実行時プロパティの関数であり得る。たとえば、各種ＩＳＰへの様々なＰＰＰ接続を区別することが望ましい場合がある。これは、ラベルをアダプタ名に割り当てる（たとえば、アダプタ「ｐｐｐ０」にラベルＬ０を割り当てる）ことによって行うことはできない。これは、アダプタ名が動的に作成され、アダプタの実際のプロパティが可変であるためである。アダプタに適切なラベルを選ぶことにより、あらゆるセキュリティチェックがラベル関数に適切に基づくことを保証することができる。これは、アダプタに付けられるラベルは、システムにすでに存在するその他のラベルに関して正しくなければならないという意味において、プロセス、ネットワーク接続、ファイル、パイプ等他のシステムオブジェクトにもラベルを付ける高信頼性オペレーティングシステム（特に、本発明者らによる第１の同時係属中の国際出願の発明の第１および第２の態様を参照して定義したもの）に関して特に重要である。
【００６５】
カーネル／オペレーティングシステムは通常、新しいアダプタがアクティブ化されるときに呼び出されるソフトウェアルーチンを有する。第２の同時係属中の国際出願の発明の例示的な実施形態では、かかるルーチンは、たとえば、規則セットまたは構成テーブルを照会することにより、新たに形成されたアダプタの属性に応じてラベルを割り当てるようにも変更される。同様に、アダプタが非アクティブ化されるときに呼び出されるルーチンもあり、これはそれまで割り当てられていたラベルを除去するように変更される。
【００６６】
本発明者らによる第１の同時係属中の国際出願の発明の第１および第２の態様を再び参照して、属するコンパートメントを示すタグで各プロセスおよびネットワークインタフェースを増補するオペレーティングシステムが定義される。例示的な実施形態では、カーネルに設けられる手段が、（任意の標準Ｕｎｉｘプロセス間通信機構を使用することで、Ｌｉ

ｎｕｘオペレーティングシステムにおいて）あるプロセスが別のプロセスと通信したいときは常に規則ベースを照会する。整合する規則が規則ベースにある場合にのみ、通信が成功する。好ましい実施形態では、規則ベースはカーネルに存在するが、上述のように、より実際的には、管理プログラムにより好ましくはユーザ空間において初期化され、動的に維持され、照会されることが望ましい。

【００６７】

したがって、本発明者らによる第１の同時係属中の国際出願の発明の第５の態様は、システムオブジェクト間の許可された通信経路を定義する１つまたは複数の規則からなる規則ベースを格納する手段を含むカーネルと、かかる規則の追加、削除、および／またはリスト化を行うユーザ操作可能手段とを備えるオペレーティングシステムを提供する。

【００６８】

したがって、本発明者らによる第１の同時係属中の国際出願の発明の第５の態様のオペレーティングシステムでは、ＴＣＰおよびＵＤＰパケットを介してのアクセス制御のみならず、オペレーティングシステムに存在する他の形態のプロセス（Ｌｉｎｕｘシステムでは、生ＩＰパケット、ＳｙｓＶメッセージ、ＳｙｓＶ共有メモリ、およびＳｙｓＶセマフォが挙げられる）間通信のアクセス制御も実行することが可能である。

【００６９】

本発明者らによる第１の同時係属中の国際出願の発明の第５の態様の例示的な一実施形態では、ユーザ空間プログラムが、規則ベース中のエントリを変更およびリスト化するために、カーネルを対象にしたデータを送受信可能である必要がある。好ましい実施形態では、これは、２つのエントリポイントを提供するカーネルデバイスドライバをオペレーティングシステムに包含することによって実施される。第１のエントリポイントは、「ｉｏｃｔｌ」システムコールのためのものである（ｉｏｃｔｌは従来、小量のデータまたはコマンドをデバイスに送信するために使用される）。第１のエントリポイントは、３つの演算に使用するよう構成される。第１に、完成した規則を指定し、規則ベースに追加するために使用することができる。第２に、同じデータを使用してその規則を削除することができる。第３に、最適化として、その「参照」により規則を削除することができ、本発明の例示的な一実施形態では、参照は、カーネルによって維持される６４ビットタグである。

【００７０】

第２のエントリポイントは、「／ｐｒｏｃ」エントリのためのものである。ユーザ空間プログラムがこのエントリを開くと、カーネルによって生成された規則のリストを読み出すことができる。この第２のエントリポイントの理由は、規則リストを読み出すには、ｉｏｃｔｌコマンドを介してよりも効率的な機構であり、また、カーネルモジュールの特定の「ｉｏｃｔｌ」コマンドを認識し処理するために、特に書く必要のない他のユーザプロセスがより容易に読み出し可能なためである。

【発明を実施するための最良の形態】

【００７１】

要約すれば、従来の高信頼性オペレーティングシステム手法と同様に、コンテインメントプロパティが、プロセス、ファイル、およびネットワーク資源をカーネルレベルで強制保護することにより、本発明の例示的な一実施形態におけるオペレーティングシステムにおいて実現される。しかし、本発明のオペレーティングシステムに使用される強制制御は、従来の高信頼性オペレーティングシステムに見られるものとはいくらか異なり、したがって、従来の高信頼性オペレーティングシステムに関連するアプリケーション統合および管理問題のいくらかを少なくとも軽減するよう意図される。

【００７２】

本発明による高信頼性オペレーティングシステムの鍵となる概念は「コンパートメント」であり、システム上の各種サービスおよびアプリケーションは別個のコンパートメント内で実行される。比較的簡単な強制アクセス制御およびプロセスラベリングを用いて、コンパートメントの概念を築く。以下の本発明による例示的な実施形態の高信頼性オペレーティングシステムでは、システム内の各プロセスにはラベルが割り振られ、同じラベルを有

するプロセスは同じコンパートメントに属する。カーネルレベルの強制チェックを実施して、あるコンパートメントのプロセスが別のコンパートメントのプロセスに干渉することができないように保証する。強制アクセス制御は、ラベルは整合するかしないかのいずれか一方であるという意味において比較的簡単である。さらに、いくつかの既知の高信頼性オペレーティングシステムでのようなラベルの階層順序付けはシステム内に存在しない。

【００７３】

従来の高信頼性オペレーティングシステムとは異なり、本発明では、メインファイルシステムへのアクセスの直接制御にラベルは使用されない。代わりに、ファイルシステム保護は、メインファイルシステムの異なるセクションに各コンパートメントを関連付けることによってなされる。ファイルシステムのかかるセクションはそれぞれ、メインファイルシステムのｃｈｒｏｏｔであり、ファイルシステムのセクションにアクセスすることができるのは、そのセクションに関連するコンパートメント内で実行中のプロセスだけである。重要なことは、カーネル制御を介して、プロセスがコンパートメント内からｒｏｏｔに移行する機能が取り除かれるため、ｃｈｒｏｏｔをエスケープすることはできない。本発明の例示的な一実施形態は、ｃｈｒｏｏｔ内の少なくとも選択されたファイルを変更不可にする機能も提供する。

【００７４】

コンパートメント間およびネットワーク資源間の柔軟性のある通信経路は、ＴＣＰ／ＵＤＰに加えて大半のＩＰＣ機構に対する狭い、カーネルレベルで制御されるインタフェースを介して設けられる。これら通信インタフェースへのアクセスは、セキュリティ管理者により「コンパートメント単位」で指定される規則によって統制される。したがって、従来の高信頼性オペレーティングシステムとは異なり、コンパートメントとネットワーク資源の間の通信を許可するために、特権を用いて強制アクセス制御を無効とするか、またはユーザレベルの高信頼性プロキシの使用にたよる必要がない。

【００７５】

したがって、本発明はコンテインメントを提供するが、アプリケーションの統合を比較的簡便にするに足る柔軟性も有し、それによって高信頼性オペレーティングシステムの配備および実行に伴う管理オーバヘッドおよび不便さを軽減する高信頼性オペレーティングシステムを提供する。

【００７６】

これより、本発明の特定の例示的な一実施形態のアーキテクチャおよび実施について述べる。以下の説明では、本発明が完全に理解されるように、多くの特定の詳細が述べられる。しかし、本発明はこういった特定の詳細に制限されることなく実施可能なことが当業者には認められよう。他の場合では、本発明を不必要に曖昧にすることを避けるために、既知の方法および構造についての詳細な説明は省く。

【００７７】

以下の説明では、ＨＴＴＰサーバ等、ユーザレベルのサービスのコンテインメントをサポートするようにベースＬｉｎｕｘカーネルを変更することによって実現される高信頼性Ｌｉｎｕｘオペレーティングシステムについて詳細に述べる。しかし、本発明の原理は、他のタイプのオペレーティングシステムにも適用することができ、同じもしくは同様の効果を上げることができることは当業者により認められよう。

【００７８】

本発明の例示的な一実施形態による高信頼性オペレーティングシステムを実現するようにＬｉｎｕｘオペレーティングシステムに加える変更は、以下のように概ね分類することができる。

【００７９】

１．以下のエリアにおけるカーネルの変更

＊ＴＣＰ／ＩＰネットワーキング

＊ルーティングテーブルおよびルーティングキャッシュ

＊システムＶ　ＩＰＣ－メッセージキュー、共有メモリ、およびセマフォ

＊プロセスおよびスレッド
＊ＵＩＤハンドリング
２．以下の形態のカーネル構成インタフェース
＊動的ロード可能カーネルモジュール
＊これらモジュールと通信するためのコマンドラインユーティリティ
３．個々のコンパートメントを管理／構成するユーザレベルのスクリプト
＊コンパートメントを開始／停止するスクリプト
図面の図２を参照して、ベースＬｉｎｕｘカーネルに対する主要な変更エリアを含み、構成可能ｃｈｒｏｏｔ　ｊａｉｌにおいてＣＧＩバイナリを実行可能なウェブサーバを実施するユーザ空間に一連のコンパートメントを追加した、本発明の例示的な一実施形態による高信頼性Ｌｉｎｕｘホストオペレーティングシステムのアーキテクチャを示す。

【００８０】
かくして、図２を参照すると、ベースＬｉｎｕｘカーネル１００は一般に、ＴＣＰ／ＩＰネットワーキング手段１０２と、Ｕｎｉｘドメインソケット１０４と、Ｓｙｓ　Ｖ　ＩＰＣ手段１０６と、他のサブシステム１０８とを備える。高信頼性Ｌｉｎｕｘオペレーティングシステムは、セキュリティモジュール１１２と、装置構成モジュール１１４と、規則データベース１１６と、カーネルモジュール１１８との形態のカーネル拡張１１０をさらに備える。図示のように、Ｌｉｎｕｘカーネルサブシステム１０２、１０４、１０６、１０８の少なくともいくつかは、カーネルレベルセキュリティモジュール１１２にコールアウトするように変更されている。セキュリティモジュール１１２は、アクセス制御判定を行い、コンパートメントの概念を実施し、それによってコンテインメントを提供する責任を有する。

【００８１】
セキュリティモジュール１１２はさらに、判定を行うときに規則データベース１１６を照会する。規則データベース１１６は、コンパートメントへまたコンパートメントから、狭く十分に制御されたインタフェースを提供するための、コンパートメント間の許容可能な通信経路についての情報を含む（図面の図１２も参照のこと）。

【００８２】
図面の図２は、カーネル拡張１１０が一連のｉｏｃｔｌコマンドを介してユーザ空間１２０からどのように管理されるかも示している。かかるｉｏｃｔｌコマンドは２つの形態をとる：規則テーブルを操作する形態、および特定のコンパートメント中のプロセスを実行し、ネットワークインタフェースを構成する形態である。

【００８３】
図２に示すウェブサーバ等のユーザ空間サービスは、プラットフォーム上で変更されずに実行されるが、セキュリティ拡張へのコマンドラインインタフェースを介してコンパートメントラベルが関連付けられている。そしてセキュリティモジュール１１２は、付けられたコンパートメントラベルに基づいて、強制アクセス制御をユーザ空間サービスに適用する責任を有する。このため、ユーザ空間サービスを変更する必要なく、こういったサービスを封じ込めることが可能なことが認められよう。

【００８４】
図面の図２を参照して述べるシステムアーキテクチャの３つの主な構成要素は、ａ）通信規則およびプロセスコンパートメントラベル等、セキュリティ拡張の根本的な態様を構成および管理するために必要なコマンドラインユーティリティ、ｂ）この機能をカーネル内で実施するロード可能モジュール、およびｃ）この機能を利用するために行われたカーネル変更である。ここで、これら３つの主な構成要素について以下により詳細に述べる。

【００８５】
ａ）コマンドラインユーティリティ
「ＣＡＣＣ」は、ｃａｃカーネルロード可能モジュール（図示せず）によって提供される／ｄｅｖ／ｃａｃｃおよび／ｐｒｏｃ／ｃａｃｃインタフェースを介して規則の追加、削除、およびリスト化を行うためのコマンドラインユーティリティである。規則は、コマン

ドラインに入力しても、またはテキストファイルから読み出してもよい。

【００８６】

本発明の例示的な本実施形態では、規則のフォーマットは以下である。

【００８７】

⟨rule⟩::=⟨source⟩[⟨port⟩]->⟨destination⟩[⟨port⟩]⟨method list⟩⟨netdev⟩

ただし、

【数１】

<identifier> == (<compartment>|<host>|<net>) [<port>]

<compartment> == 'COMPARTMENT' <comp_name>

<host> == 'HOST' <host_name>

<net> == 'NET' <ip_addr> <netmask>

<net> == 'NET' <ip_addr> '/' <bits>

【００８８】

⟨comp_name⟩==コンパートメントの有効な名前

⟨host_name⟩==既知のホスト名またはＩＰアドレス

⟨ip_addr⟩==a.b.c.d形態のＩＰアドレス

⟨netmask⟩== a.b.c.d形態の有効ネットマスク

⟨bits⟩==ネットマスクにおける最左端ビットの数字、０～３１

⟨method_list⟩==カンマで区切られたメソッドのリスト（例示的な本実施形態において、サポートされるメソッドは、ＴＣＰ（伝送制御プロトコル）、ＵＤＰ（ユーザデータグラムプロトコル）、およびＡＬＬである。

【００８９】

規則を追加するには、ユーザは「ｃａｃｃ－ａ＜ｆｉｌｅｎａｍｅ＞」を入力する（＜ｆｉｌｅｎａｍｅ＞が上記フォーマットの規則を含むファイルである場合、テキストファイルから規則を読み出す）か、または「ｃａｃｃ－ａ　ｒｕｌｅ」を入力する（規則をコマンドラインに入力する）ことができる。

【００９０】

規則を削除するには、ユーザは、「ｃａｃｃ－ｄ＜ｆｉｌｅｎａｍｅ＞」、またはｃａｃｃ－ｄ　ｒｕｌｅ、もしくはｃａｃｃ－ｄ　ｒｅｆ（この形では、単に、コマンドｃａｃｃ－ｌを使用して規則をリスト化することによって出力される参照番号によって規則を削除することができ、コマンドｃａｃｃ－ｌは、標準形式で規則を出力またはリスト化し、規則参照は各規則の終わりに注釈として出力される）を入力することができる。

【００９１】

デフォルトにより、「ｃａｃｃ」はカレント作業ディレクトリにおいてコンパートメントマッピングファイル「ｃｍａｐ．ｔｘｔ」およびメソッドマッピングファイル「ｍｍａｐ．ｔｘｔ」を見つけるものと期待される。しかしこれは、本発明の例示的な本実施形態では、ＵＮＩＸ環境変数ＣＡＣＣ＿ＣＭＡＰおよびＣＡＣＣ＿ＭＭＡＰを実際にファイルが常駐するところにセットすることによって無効とされる可能性がある。

【００９２】

ｃａｃｃによって捕捉されたあらゆる構文エラーまたは意味エラーは、エラーレポートをもたらし、コマンドが即時終了し、規則が追加または削除されないことになる。テキストファイルが規則の入力に使用されている場合には、エラーのあるラインのライン番号がエ

ラーメッセージに提供される。
【００９３】
本発明の例示的な本実施形態によって提供される別のコマンドラインユーティリティは、「ｌｃｕ」として知られており、ＬＮＳカーネルモジュール（図示せず）へのインタフェースを提供する。ｌｃｕの最も重要な機能は、各種管理スクリプトに、所与のコンパートメント中のプロセスを生成する機能およびインタフェースのコンパートメント数をセットする機能を提供することである。以下は使用例である。
【００９４】
１．'lcu setdev eth0 0xFFFF0000'
ｅｔｈ０ネットワークインタフェースのコンパートメント番号を０ｘＦＦＦＦ０００００にセットする
２．'lcu setprc 0x2-cap_mknod bash'
コンパートメント０ｘ２に切り換え、ｃａｐ＿ｍｋｎｏｄ機能を解除し、ｂａｓｈを呼び出す
ｂ）カーネルモジュール
本発明の例示的な本実施形態は、カスタムｉｏｃｔｌ（　）の実施に、規則の挿入／削除およびネットワークインタフェースのラベル付け等他の機能を可能にする２つのカーネルモジュールを使用する。しかし、２つのモジュールをカスタムシステムコールに併合し、かつ／またはカスタムシステムコールで置き換えてもよいものと考えられる。本発明の本実施形態では、２つのカーネルモジュールの名前はｌｎｓおよびｃａｃである。
【００９５】
ｌｎｓモジュールは、カスタムｉｏｃｔｌ（　）を介して各種インタフェースを実施し、以下を可能にする。
【００９６】
１．呼び出しプロセスがコンパートメントを切り換えること、
２．個々のネットワークインタフェースにコンパートメント番号を割り当てること。
【００９７】
コンパートメント番号を使用してのプロセスリスト化、およびカーネルレベルのセキュリティチェックに対するアクティビティのログ化等のユーティリティ関数。
【００９８】
このモジュールの主なクライアントは、上述したｌｃｕコマンドラインユーティリティである。
【００９９】
ｃａｃモジュールは、インタフェースを実施して、カスタムｉｏｃｔｌ（　）を介してカーネルにおいて規則を追加／削除する。これは、より上位レベルの簡略化された規則を、カーネル参照ルーチンがより理解し易い原始的な形に翻訳する。このモジュールは、ｃａｃｃおよびｃｇｉｃａｃｃユーザレベルユーティリティと呼ばれ、カーネル内の規則を操作する。
【０１００】
ｃ）カーネル変更
本発明の例示的な本実施形態では、各種データ型に付けられるタグを導入し、このようにタグ付けされたデータ型に対してもアクセス制御チェックをさらに行うように、標準Ｌｉｎｕｘカーネルソースに変更が加えられている。タグ付けされた各データ型は追加として、コンパートメント番号を保持するために用いられるｓｔｒｕｃｔ　ｃｓｅｃｉｎｆｏデータメンバ（図面の図３に示す）を含む。タグ付けされたデータ型は、他のセキュリティ属性も保持するように拡張可能なものと考えられる。一般に、このデータメンバの追加は通常、共通のエントリで始まる２つまたはそれよりも多くの名前の異なる構造体にポインタを向ける慣行に関連して発生する問題を回避するために、データ構造体の最後の最後で行われる。
【０１０１】

個々のカーネル資源をタグ付けする最終的な効果は、生成／消費するプロセスおよびデータが相互に分離されたコンパートメント化システムの実施が非常に簡単なことである。かかる分離は、多くの隠れチャネルが存在する（プロセスについての以下の考察を参照）という意味において、厳密な分離を意図しない。分離は単に、論理的に異なるプロセス群の間の明確な形のコンフリクトおよび／または相互作用を回避するものと意図される。

【０１０２】

本発明の例示的な本実施形態では、カーネルにおいて保護されているサブシステムにイエス／ノーセキュリティチェックを実施する単一関数ｃｎｅｔ＿ｃｈｋ＿ａｔｔｒ（　）が存在する。この関数への呼び出しは、カーネルソースにおいて、要求されたコンパートメント化動作を実施するに相応しいポイントで行われる。この関数は、考慮するサブシステムに基づき、そのときに照会されているオペレーションがあるサブシステムに応じてわずかに異なるデフォルトまたは規則規定を実施することができる。たとえば、大半のサブシステムは単純なパーテーション化を実施し、まったく同じコンパートメント番号を有するオブジェクト／資源にしか肯定の値が返されない。しかし、特定の場合では、非特権コンパートメント０および／またはワイルドカードコンパートメント－１Ｌの使用を用いることができ、たとえば、コンパートメント０を分類されていない資源／サービスのデフォルト「サンドボックス」として使用し、ワイルドカードコンパートメントを、シャットダウン前にサブシステム上のすべてのプロセスをリスト化するなど、管理目的に使用することができる。

【０１０３】

図面の図４を参照して、標準ＬｉｎｕｘＩＰネットワーキングについてまず説明する。各プロセスまたはスレッドは、カーネル中のｔａｓｋ＿ｓｔｒｕｃｔ変数で表される。プロセスは、ＴＣＰ／ＵＤＰを介して、ネットワーク通信のためのＡＦ＿ＩＮＥＴドメインにソケットを作成する。これらは、これもまたカーネル中のｓｔｒｕｃｔソケットおよびｓｔｒｕｃｔ　ｓｏｃｋ変数の対で表される。

【０１０４】

ｓｔｒｕｃｔ　ｓｏｃｋデータ型は、とりわけ、ｓｔｒｕｃｔ　ｓｋ＿ｂｕｆｆで表される入力パケットのキューを含む。これはまた、パケット伝送用の予め割り当てられたｓｋ＿ｂｕｆｆのキューも保持する。各ｓｋ＿ｂｕｆｆは、ＩＰスタックの上下に移行するＩＰパケットおよび／またはフラグメントを表す。ｓｔｒｕｃｔ　ｓｏｃｋから（または、より具体的には、内部の予め割り当てられた送信キューから）発せられ、伝送のために下方に移行するか、あるいはネットワークドライバから発せられ、ネットワークインタフェースを表すｓｔｒｕｃｔ　ｎｅｔ＿ｄｅｖｉｃｅから始まるスタックの底から上方に移行する。下方に移行する場合には、ｓｔｒｕｃｔ　ｎｅｔ＿ｄｅｖｉｃｅで効果的に終わる。上方に移行する場合には、通常、待機中のｓｔｒｕｃｔ　ｓｏｃｋ（実際、その保留キュー）に送られる。

【０１０５】

ｓｔｒｕｃｔ　ｓｏｃｋ変数は、ｓｏｃｋｅｔ（　）コールによって本質的に間接的に作成され（実際には、実行中のプロセスまでトレースすることができないカーネル自体内のスタックの各種部分が所有するプロトコル単位のプライベートソケットがある）、通常、所有するユーザプロセス、すなわちｔａｓｋ＿ｓｔｒｕｃｔまでトレースすることができる。ｓｔｒｕｃｔ　ｎｅｔ＿ｄｅｖｉｃｅ変数は、ループバックインタフェースを含む、システム上の各構成インタフェースに存在する。ローカルホストおよびループバック通信は、速度のためにスタックを横切る高速パスを介して移行するようには見えず、代わりに、リモートホスト通信に予想されるようにスタックの上下に移行する。スタック中の各種ポイントにおいて、パケットインタセプトの目的で、登録されたネットフィルタモジュールにコールすることができる。

【０１０６】

追加のｃｓｅｃｉｎｆｏデータメンバを、ＬｉｎｕｘＩＰネットワーキングにおいて最も一般に使用されるデータ型に追加することにより、カーネル生成応答を含め、システム上

で実行中のすべてのプロセスについて、個々のＩＰパケットの所有権、ひいては読み出し／書き込みデータフローのトレースが可能になる。

【０１０７】

したがって、本発明の例示的な本実施形態を容易にするように、標準ＬｉｎｕｘＩＰネットワーキングに使用される少なくとも主要なネットワーキングデータ型を変更した。実際には、本発明の本実施形態を実現するために変更されたデータ構造体の大半は、ネットワーキングに関連するものであり、ネットワーキングスタックおよびソケットサポートルーチンに存在する。タグ付けされたネットワークデータ構造体は、区分けされたＩＰスタックの実施に役立つ。本発明の例示的な本実施形態では、ｓｔｒｕｃｔ　ｃｓｅｃｉｎｆｏを包含するように以下のデータ構造体を変更した。

【０１０８】

１．struct task_struct－プロセス（およびスレッド）
２．struct socket－抽象ソケット表現
３．struct sock－ドメイン固有ソケット
４．struct sk_buff－ソケット間のＩＰパケットまたはメッセージ
５．struct net_device－ネットワークインタフェース、たとえばｅｔｈ０　ｌｏ等

セットアップ中、主要なデータ型がタグ付けされると、これらデータ型が新たに初期化された変数をカーネルに導入するように使用されたポイントについて、ＩＰスタック全体がチェックされた。いったん、かかるポイントが識別されると、ｃｓｅｃｉｎｆｏ構造体の継承が確実に実行されるようにコードが挿入された。ｃｓｅｃｉｎｆｏ構造体がＩＰネットワーキングスタックを通して伝播する様式について、より詳細に次に述べる。

【０１０９】

ｓｔｒｕｃｔ　ｃｓｅｃｉｎｆｏデータメンバには、名前の付いた２つのソース、すなわちプロセス単位のｔａｓｋ＿ｓｔｒｕｃｔおよびインタフェース単位のｎｅｔ＿ｄｅｖｉｃｅがある。各プロセスは、特権ｉｏｃｔｌ（　）により明示的に変更されない限り、親からｃｓｅｃｉｎｆｏを継承する。本発明の例示的な本実施形態では、ｉｎｉｔ－ｐｒｏｃｅｓｓにコンパートメント番号０が割り当てられる。そのため、システムスタートアップ中にｉｎｉｔにより生成されるあらゆるプロセスは、明示的に別にセットされる場合を除き、このコンパートメント番号を継承する。システムスタートアップ中、ｉｎｉｔ－ｓｃｒｉｐｔが通常呼び出されて、定義された各ネットワークインタフェースにコンパートメント番号をセットする。図面の図５は、最も一般的な場合に、ｃｓｅｃｉｎｆｏデータメンバがどのように伝播するかを示している。

【０１１０】

他のデータ構造体はすべて、それぞれのｃｓｅｃｉｎｆｏ構造体をｔａｓｋ＿ｓｔｒｕｃｔから、あるいはｎｅｔ＿ｄｅｖｉｃｅから継承する。たとえば、プロセスがソケットを作成する場合、呼び出しプロセスからカレントｃｓｅｃｉｎｆｏを継承するｓｔｒｕｃｔ　ｓｏｃｋｅｔおよび／またはｓｔｒｕｃｔ　ｓｏｃｋが作成され得る。ｗｒｉｔｅ（）をソケットに呼び出すことによって続けて生成されるパケットは、それぞれのｃｓｅｃｉｎｆｏを発信元ソケットから継承するｓｋ＿ｂｕｆｆを生成する。

【０１１１】

入力ＩＰパケットには、到着したネットワークインタフェースのコンパートメント番号がスタンプされるため、スタックを上方に移行するｓｋ＿ｂｕｆｆは、それぞれのｃｓｅｃｉｎｆｏ構造体を発信元ｎｅｔ＿ｄｅｖｉｃｅから継承する。ソケットに送られる前に、各ｓｋ＿ｂｕｆｆのｃｓｅｃｉｎｆｏ構造体が、期待されるソケットのｃｓｅｃｉｎｆｏ構造体と照らし合わせてチェックされる。

【０１１２】

非リモートネットワーキングの場合、すなわち接続が、以下の形態の規則によって許可される複数のネットワークインタフェースのいずれか１つを通してコンパートメントＸとＹの間が接続される場合には、特別な注意を払わなければならないことは認められよう。

【０１１３】

COMPARTMENT X->COMPARTMENT Y METHOD tcp

セキュリティチェックはＩＰネットワーキングには二度、すなわち出力に対して一度、および入力に対して一度行われるため、システムが代わりにこれら規則の存在を探すことを阻止する手段を提供する必要がある。

【０１１４】

COMPARTMENT X->HOST a.b.c.d METHOD tcp（出力の場合）

HOST a.b.c.d->COMPARTMENT Y METHOD tcp（入力の場合）

これは有効であるが、発信元および宛先のコンパートメントを直接指定する規則に優先しては使用されない場合がある。これを考慮に入れるために、本発明の例示的な本実施形態では、ループバックデバイスに送られるパケットは、それぞれ元のコンパートメント番号を保持し、最終的な送信に単に「反映」させる。この場合、セキュリティチェックは引き渡しに対して行われ、伝送に対しては行われないことに留意する。入力ローカルパケットをループバックインタフェース上で受け取ると、システムはセットアップされ、パケットのコンパートメント番号がネットワークインタフェースのコンパートメント番号で上書きされず、引き渡しに対する最終的なチェックのためにスタックの上方に移行させる。いったん上方に移行すると、システムは、

HOST a.b.c.d->COMPARTMENT Y METHOD tcpの代わりに、

COMPARTMENT X->COMPARTMENT Y tcp

の形態の規則をチェックする。これは、ネットワークインタフェース（本発明の例示的な本実施形態におけるネットワークインタフェースは、一般規則として、０ｘＦＦＦＦ００００よりも上の範囲のコンパートメント番号が割り当てられるため、実行中のサービスに割り当てられるコンパートメント番号と区別することができる）に通常割り当てられる形態ではないコンパートメント番号がｓｋ＿ｂｕｆｆに存在するためである。

【０１１５】

規則は一方向性のものであるため、ＴＣＰレイヤは、ｃｏｎｎｅｃｔ（　）あるいはａｃｃｅｐｔ（　）の結果としてＴＣＰ接続がセットアップされると、逆データフローを扱う規則を動的に挿入する必要がある。これは、本発明の例示的な本実施形態では自動的に行われ、ＴＣＰ接続が閉じられると、規則は削除される。ｓｔｒｕｃｔ　ｓｏｃｋの形態で完全にセットアップされたものとは対照的に、ｓｔｒｕｃｔ　ｔｃｐ＿ｏｐｅｎｒｅｑが作成され保留中の接続要求の状態を表す場合には特別な処理が行われる。作成されたリバース規則の参照は、保留中要求とともに格納され、接続要求がタイムアウトするか、ある他の理由により失敗した場合にも削除される。

【０１１６】

この例は、コンパートメント２からリモートホスト１０．１．１．１に接続が行われる場合である。かかる動作を許す元の規則は、以下のようなものであり得る。

【０１１７】

COMPARTMENT 2->NET 10.1.1.0/255.255.255.0 METHOD tcp

その結果、リバース規則はこのようなものである（abc/xyzが使用される特定のポート番号）。

【０１１８】

HOST 10.1.1.1 PORT abc->COMPARTMENT 2PORT xyz METHOD tcp

コンパートメント単位のルーティングテーブルをサポートするために、各ルーティングテーブルエントリにｃｓｅｃｉｎｆｏ構造体がタグ付けされる。本発明の例示的な実施形態における、変更された各種データ構造体は以下である。

【０１１９】

１．ｓｔｒｕｃｔ ｒｔ_ｋｅｙ
２．ｓｔｒｕｃｔ ｒｔａｂｌｅ
３．ｓｔｒｕｃｔ ｆｉｂ_ｒｕｌｅ
４．ｓｔｒｕｃｔ ｆｉｂ_ｎｏｄｅ
ルートコマンドを使用してルートを挿入すると、ユーザプロセスの呼び出しコンテキストから受け継いだｃｓｅｃｉｎｆｏ構造体を有するルーティングテーブルエントリが挿入される。すなわち、ユーザがコンパートメントＮ中のシェルからルートコマンドを呼び出すと、追加されるルートに、コンパートメント番号としてＮがタグ付けされる。ルーティングテーブル情報を見ようという試み（通常、／ｐｒｏｃ／ｎｅｔ／ｒｏｕｔｅおよび／ｐｒｏｃ／ｎｅｔ／ｒｔ＿ｃａｃｈｅを調べることにより）は、呼び出しユーザプロセスのｃｓｅｃｉｎｆｏ構造体の値に基づく。

【０１２０】
ｓｋ＿ｂｕｆｆがとるべき入出力ルートの判定に使用される主なルーチンは、ｉｐ＿ｒｏｕｔｅ＿ｏｕｔｐｕｔ（ ）およびｉｐ＿ｒｏｕｔｅ＿ｉｎｐｕｔ（ ）である。本発明の例示的な本実施形態では、これらのルーチンが、あらゆるルーティングテーブル参照の土台であるｃｓｅｃｉｎｆｏ構造体へのポインタからなる追加の引数を含むように拡張されている。追加されるこの引数は、入力あるいは出力のためにルーティングされているパケットのどちらか一方のｓｋ＿ｂｕｆｆから与えられる。

【０１２１】
カーネルに挿入されたルーティングエントリは、特別な状態を有し、ワイルドカードコンパートメント番号（－１Ｌ）が挿入される。コンパートメント単位ルーティングの状況では、すべてのコンパートメントがこういったエントリを共有することができる。かかる機構の主な目的は、入力パケットをスタックに適宜ルーティング可能にすることである。セキュリティチェックはいずれも、ｓｋ＿ｂｕｆｆがソケット（またはｓｋ＿ｂｕｆｆキュー）で送られる直前に、より上位のレベルで行われる。

【０１２２】
最終的な効果は、各コンパートメントが、デフォルトでは空のルーティングテーブルをそれぞれ個々に有するように見えることである。あらゆるコンパートメントは、システム全体のネットワークインタフェースを共用する。本発明の例示的な本実施形態では、個々のコンパートメントを、利用可能なネットワークインタフェースの限られた部分集合に制限することが可能である。これは、各ネットワークインタフェースが概念的にそれぞれのコンパートメントにある（それぞれのルーティングテーブルとともに）ためである。実際、ＩＣＭＰエコー要求に応答して、個々の各インタフェースは、オプションとして、ルーティングテーブルエントリをタグ付けして、プロトコル単位のＩＣＭＰソケットを出力パケットにルーティング可能にするように構成することができる。

【０１２３】
他のサブシステム
＊ＵＮＩＸドメインソケット－各ＵＮＩＸドメインソケットもまた、ｃｓｅｃｉｎｆｏ構造体がタグ付けされる。これらもｓｋ＿ｂｕｆｆを使用して、接続されたソケット間を移行するメッセージ／データを表すため、上述したＡＦ＿ＩＮＥＴドメインが使用する機構の多くも同様に適合する。加えて、セキュリティチェックもまた、ピアに接続する試みがある都度行われる。

【０１２４】
＊システムＶ　ＩＰＣ－上に列挙した各ＩＰＣ機構は、同様にｃｓｅｃｉｎｆｏ構造体がタグ付けされた専用カーネル構造体を使用して実施される。これら構造体へのメッセージのリスト化、追加または除去を行おうとする試みは、個々のｓｋ＿ｂｕｆｆと同じセキュリティチェックの対象となる。セキュリティチェックは、使用される機構の厳密なタイプに依存する。

【０１２５】
＊プロセス／スレッド－個々のプロセス、すなわちｔａｓｋ＿ｓｔｒｕｃｔにはｃｓｅｃ

ｉｎｆｏ構造体がタグ付けられるため、大半のプロセス関連演算は、プロセスのコンパートメント番号の値に基づくことになる。特に、プロセスリスト化（／ｐｒｏｃインタフェースを介して）は、コンパートメント単位のプロセスリスト化の効果を上げるように制御される。信号送出は、コンパートメントが切り換えられ、そのために１ビット隠れチャネルを構成している場合もある親プロセスへの信号送出に関して考慮すべき問題があるため、いくらか複雑である。

【０１２６】

システムデフォルト

プロトコル単位のソケット－ＬｉｎｕｘＩＰスタックは、特別なプライベートプロトコル毎ソケットを使用して、ＩＣＭＰ応答等デフォルトの各種ネットワーキングの振る舞いを実施する。これらプロトコル毎のソケットは、どのようなユーザレベルソケットにも限定されず、通常、ワイルドカードコンパートメント番号で初期化され、ネットワーキング関数が通常通り振る舞えるようにする。

【０１２７】

コンパートメント０の非特権デフォルトとしての使用－規定は、コンパートメント０による他のコンパートメントおよびネットワーク資源へのアクセスを許可する規則をいずれも決して挿入しないというものである。このように、初期化されたオブジェクト、または適宜説明されていないオブジェクトのデフォルトの振る舞いは、賢明で限定的なデフォルトに分類される。

【０１２８】

デフォルトカーネルスレッド－各種カーネルスレッド、たとえば、数例を挙げればｋｓｗａｐｄ、ｋｆｌｕｓｈｄ、およびｋｕｐｄａｔｅがデフォルトで現れ得る。これらスレッドにもｔａｓｋ＿ｓｔｒｕｃｔ毎にｃｓｅｃｉｎｆｏ構造体が割り当てられ、それぞれのコンパートメント番号は、それぞれの相対的な非特権状態を反映してデフォルトで０である。

【０１２９】

ｒｏｏｔＩＤの奪取に対抗してのコンパートメント封印－個々のコンパートメントは、オプションとして、そのコンパートメント中のプロセスがｓｅｔｕｉｄ（０）および友人（ｆｒｉｅｎｄｓ）の呼び出しが成功しないように、またいかなるＳＵＩＤ－ｒｏｏｔバイナリも実行されないように「封印」と登録することができる。これは通常、一般に悪意のあるコードの実行につながるバッファオーバーフロー攻撃を受けやすい場合がある外部アクセス可能なサービスに使用される。かかるサービスが最初に擬似ユーザ（非ｒｏｏｔ）として実行されるように制限され、また中で実行するコンパートメントが封印されている場合、バッファオーバーフロー攻撃および／または外部命令の実行によりｒｏｏｔアイデンティティを奪取する試みはいずれも失敗する。なお、ｒｏｏｔとして実行中の既存プロセスは、いずれも実行し続ける。

【０１３０】

上に述べたカーネル変更は、保護されたコンパートメント中の個々のユーザレベルサービスのホスティングのサポートに役立つ。これに加えて、本発明の例示的な本実施形態においてサービスを追加または除去する際に使用されるレイアウト、ロケーション、および規定についてこれより述べる。

【０１３１】

個々のサービスには一般にそれぞれコンパートメントが割り当てられる。しかし、エンドユーザがサービスとして認めるものは、実際には結局いくつかのコンパートメントを使用することになる場合がある。例は、それぞれ個々のコンパートメントにおいてＣＧＩバイナリを実行する高信頼性ゲートウェイエージェントをホストする別のコンパートメントに対する狭いインタフェースを備えた外部アクセス可能なウェブサーバをホストするコンパートメントの使用である。この場合、少なくとも３つのコンパートメントが必要である。

【０１３２】

＊ウェブサーバプロセス用のコンパートメント

＊ＣＧＩバイナリを実行する高信頼性ゲートウェイエージェント用のコンパートメント
＊高信頼性ゲートウェイはそれぞれの構成コンパートメントにおいてＣＧＩバイナリをフォーク／実行するため、ＣＧＩバイナリの適宜分類に必要な数のコンパートメント
あらゆるコンパートメントは名前を有し、／ｃｏｍｐｔの下でｃｈｒｏｏｔ可能な環境として常駐する。本発明の例示的な実施形態において使用する例は、以下を含む。
【表１】

ロケーション	説明
/compt/admin	ａｄｍｉｎ　ＨＴＴＰサーバ
/compt/omailout	ＯｐｅｎＭａｉｌサーバプロセスをホストする、外部から見えるＨＴＴＰサーバ
/compt/omailin	ＯｐｅｎＭａｉｌサーバプロセスをホストする内部コンパートメント
/compt/web1	外部から見えるＨＴＴＰサーバ
/compt/web1mcga	ウェブ１のＣＧＩバイナリ用の内部高信頼性ゲートウェイエージェント

【０１３３】
加えて、以下のサブディレクトリも存在する。
【０１３４】
１．/compt/etc/cac/bin－コンパートメント管理用の各種スクリプトおよびコマンドラインユーティリティ
２．/compt/etc/cac/rules－システム上の登録されたあらゆるコンパートメントの規則を含むファイル
３．/compt/etc/cac/encoding－ｃａｃｃユーティリティ、たとえばコンパートメント名マッピングの構成ファイル
コンパートメントの総称的な開始／停止をサポートするために、各コンパートメントは少数の基本的な要件を満たす必要がある。
【０１３５】
１．コンパートメントロケーション／ｃｏｍｐｔ／＜ｎａｍｅ＞下でｃｈｒｏｏｔ可能であること。
【０１３６】
２．コンパートメントの開始／停止に／ｃｏｍｐｔ／＜ｎａｍｅ＞／ｓｔａｒｔｕｐおよび／ｃｏｍｐｔ／＜ｎａｍｅ＞／ｓｈｕｔｄｏｗｎを提供すること。
【０１３７】
３．スタートアップおよびシャットダウンスクリプトが、規則挿入、ルーティングテーブル作成、ファイルシステム搭載（たとえば、／ｐｒｏｃ）、および他のサービス毎の初期化ステップに対する責任を有すること。
【０１３８】
一般に、コンパートメントを外部から見えるようにすべき場合、そのコンパートメント中のプロセスは、デフォルトによりｒｏｏｔとして実行すべきではなく、コンパートメントは初期化後に封印すべきである。統合／移植されているレガシーアプリケーションの性質により、これが可能ではない場合もあり、その場合には、プロセスがｃｈｒｏｏｔ－ｊａｉｌ、たとえばｃａｐ＿ｍｋｎｏｄをエスケープしないように、可能な限り多くの機能を除去することが望ましい。
【０１３９】
各種管理スクリプトは各構成コンパートメントのファイルシステムにアクセスする必要があり、またこれら管理スクリプトは、管理ウェブサーバのＣＧＩインタフェースを介して呼び出されることから、これらスクリプトは正常なコンパートメントとして、すなわち／ｃｏｍｐｔ／＜ｎａｍｅ＞下に常駐することができないと言える。

【０１４０】
本発明の例示的な本実施形態では、使用される手法は、あらゆる構成コンパートメントの管理スクリプトのｃｈｒｏｏｔ可能な環境を封じ込めるが、その環境が確実にホストファイルシステムの限られた部分集合であるようにするというものである。自然な選択としては、管理スクリプトのｃｈｒｏｏｔ－ｊａｉｌにｒｏｏｔ　ａｔ／ｃｏｍｐｔを持たせる。結果得られる構造を図面の図１１に模式的に示す。
【０１４１】
コンパートメントは、／ｃｏｍｐディレクトリ下のｃｈｒｏｏｔ化環境として存在するため、アプリケーションの統合に必要なのは、確実にｃｈｒｏｏｔ化環境で働くようにするために使用される普通の技法である。一般的な技法では、インストールされたソフトウェアの最小ＲＰＭデータベースを含む、最小実行中のコンパートメントのｃｐｉｏ－ａｒｃｈｉｖｅを用意する。所望のアプリケーションをこのトップにインストールすることが普通であり、ＲＰＭの形態のアプリケーションの場合、以下のステップを行うことができる。
【０１４２】
root@tlinux# chroot/compt/appl
root@tlinux# rpm-install(PRM-package-filename)
root@tlinux# [必要に応じて構成ファイル、たとえばｈｔｔｐｄ．ｃｏｎｆを変更]
root@tlinux# [スタートアップ／シャットダウンスクリプトを／ｃｏｍｐｔ／ａｐｐｌに作成]

後の少数のステップは、ＲＰＭインストール段階に統合することができる。ディスクスペースの削減は、検査、すなわちｒｐｍコマンドを介して使用されていないパッケージを選択的にアンインストールすることによって実現することができる。必要であれば、コンパートメントの／ｄｅｖディレクトリ中にエントリをさらに作成することができるが、通常、／ｄｅｖは殆どの場合、実質的にありのままである。さらなる自動化は、ウェブベースのインタフェースを上記プロセスに提供して、インストールするアプリケーションの各タイプに必要なパラメータをすべて与えることによって実現することができる。かかるアプリケーションのコンパートメントアウェアバリアントをインストールする必要がある場合を除いて、コンパイルされたバイナリを変更する必要は一般にない。
【０１４３】
本発明の一態様の特定の実施形態について詳細に上述した。しかし、多種多様な異なる技法を、本発明によって提供されるコンテインメントの一般的な概念の実施態様に使用することができる。オペレーティングシステムの書き換えは、可能な限り多くのユーザレベルアプリケーションを再使用可能である必要があるため、明らかに望ましくない。このため介在技法（interposition technique）があり、そのうちのいくつかを以下に列挙する。介在技法は、主にユーザレベルで動作しているもの、あるいはカーネルベースで動作しているものに分類することができる。
【０１４４】
ユーザレベルの技法
以下に、３つの一般的なユーザレベルの技法または機構について概説する。
【０１４５】
１．ｓｔｒａｃｅ（　）機構
この機構は、システムカーネルに構築された機能を使用して、選択されたプロセスの各システムコールをトレースする。この機構を使用すると、各システムコールおよびその引数を識別することができ、システムコールは通常、規定されたセキュリティポリシーに従って続行許可されるか（引数が変更されることがある）、あるいは失敗する。
【０１４６】

この機構は、多くのアプリケーションに適しているが、いくつかの欠点を有する。これら欠点の１つは、トレースされているプロセスＰが、ｆｏｒｋ（　）システムコールから戻る前に実行がスケジューリングされた子Ｑをフォークする可能性がある「子の暴走（runaway child）」問題の場合に明らかになる。ｓｔｒａｃｅ（　）は、プロセスＩＤ（ＰＩＤ）を使用してプロセスに添付することによって働き、ＱのＰＩＤは、Ｑが実際に実行をスケジューリングされる前にＰ（ひいてはトレーサ）に必ずしも戻されるわけではないため、トレーサを添付することができるようになる前に、Ｑが任意の長さのコードの実行を許可されてしまう危険性がある。

【０１４７】

この問題に対する１つの方策は、まだトレースされていないプロセスについてカーネル中のあらゆるシステムコールをチェックし、トレーサが最終的に追いつくことができるように、たとえば、まだトレースされていないプロセスを強制的に「スリープさせる」ことにより、その場所に留めることである。しかし、この方策では、追加のカーネルコンポーネントが必要である。

【０１４８】

２．システムコールラッピング

この機構の別の欠点は、トレースされたシステムコールへの引数が変更されることがある競合状況が存在する場合に発生する。これが発生する期間は、トレーサが引数のセットを検査するときから、実際にシステムコールの続行を許可するときまでである。トレースされたプロセスと同じアドレス空間を共有するスレッドは、この間隔中にメモリにおける引数を変更することができる。

【０１４９】

この機構を使用すると、トレースする必要のあるプロセスに対してリンクされた、システムコールへのラッパを含む動的リンク共有ライブラリを用いてシステムコールをラップすることができる。こういったラッパは、予め定義されたセキュリティポリシーに従って判定を行うモジュールへのコールアウトを含むことができる。

【０１５０】

この機構に関連する１つの欠点は、プロセスが使用するものと想定されるシステムコールが未解決の外部参照ではなく、動的ローダによりリンクすることができない場合に容易に破壊される可能性があることである。また、レジスタのセットアップが通常のシステムコールのように正しい状態で、プロセスがソフト割り込み自体を実行する場合、システムコールにラッパを迂回させることも可能である。この場合、カーネルは、ラッパに渡すことなくコールを処理する。加えて、場合によっては、ＬＤ＿ＰＲＥＬＯＡＤ環境変数への依存もまた、許容できない弱いリンクであることもある。

【０１５１】

３．ユーザレベルの認証サーバ

このカテゴリは、カーネルへのプライベートチャネルを介して与えられるデータに対して作用するユーザ空間における認証サーバを含む。この手法は、多くの場合に非常に効果的であるが、いくつかの欠点を有する。すなわち、Ｉ）チェック中の各システムコールが少なくとも２つのコンテキスト切り換えを受け、この方策が比較的低速になること、ｉｉ）割り込みルーチンが、スリープしないという要件によりユーザ空間カーネルにブリッジすることがより難しいこと、およびｉｉｉ）カーネルレベルのコンポーネントが通常、強制トレースを実施する必要があること、である。

【０１５２】

上に概説したユーザレベル手法の欠点にも関わらず、本発明の一態様による高信頼性オペレーティングシステムを実施するユーザレベル技法は、開発および維持が比較的容易であるという利点を有するが、状況によっては、システム全体の強制制御の実施には不十分なことがある。

【０１５３】

最終的に、本発明の目的は、実行中のアプリケーションを封じ込めることであり、これは

好ましくは、セキュリティ管理者により直接許可されていないエージェントが自由裁量ベースで無効とすることができない一連の強制アクセス制御によって実施される。実行中の第三者アプリケーションにトランスペアレントなコンテインメントの実施は、カーネルレベルのアクセス制御によって実現することができる。考えられるエントリポイントを調べ、互いの中および互いに対するカーネルサブシステムの相互作用を分離することにより、カーネルおよびその資源のビューを実行中のアプリケーションに関してセグメント化することが可能になる。

【０１５４】

かかるセグメント化方式は、カーネル自体内での実施により強制的な性質のものであり、コンテインメント方式を明らかに承知して、それを利用するように書き換えられた場合を除き、実行中のアプリケーションによって無効とされる可能性がある自由裁量的な側面がない。

【０１５５】

本発明を実施するカーネルレベルの手法の３つの例を以下に概説し、図面の図６に示す。第１の手法は主に、カーネルおよびその内部データ構造へのパッチに基づく。第２の手法は、いずれのカーネルパッチもまったく必要としないという点においてまったく異なり、代わりに、選択されたシステムコールを置き換え、おそらく実行時のカーネルのイメージを変更することによって動作する動的ロード可能カーネルモジュールである。これら手法は両方とも、通常、カーネルへのプライベートチャネルを介して動作するユーザレベルの構成ユーティリティを必要とする。第３の手法は、第１の手法によって提供される絶対制御と、第２の手法によって提供されるカーネルソース変更からの独立とを折衷したものを表す。

【０１５６】

１．コンテインメントをサポートするようにするソースレベルでのカーネル変更（Ｖ１）

この手法は、標準オペレーティングシステム（この場合、Ｌｉｎｕｘ）のカーネルソースへの一連のパッチとして実施される。規則テーブルの維持に必要なロジックをホストし、カーネルとユーザ空間構成ユーティリティとの間のインタフェースとしても動作する動的ロード可能カーネルモジュールもある。カーネルモジュールは、初期にブートシーケンスに挿入され、定義された規則がない状態で限定セキュリティモデルを即座に実施する。これに先立って、カーネルは、すべてのプロセスが、機能可能であるが、本質的に大半の目的に役立たないデフォルトコンパートメント０で生成される状態で、適切にブートできるように設計された限定セキュリティモデルを実施する。カーネルモジュールがロードされると、カーネルは内蔵モデルからモジュールにおけるモデルに切り替わる。コンテインメントは、カーネル資源をタグ付けし、タグの値および規定されているかもしれないあらゆる規則に応じてこれら資源へのアクセスを分割することによって実現される。

【０１５７】

したがって、保護が必要な各カーネル資源は、資源が属するコンパートメントを示すタグで拡張される（上に述べたように）。コンパートメントは、カーネル内の単一ワードサイズの値によって表されるが、ユーザレベル構成ユーティリティはより記述的なストリング名を使用する。かかる資源の例としては、以下を記述するデータ構造体が挙げられる。

【０１５８】

＊個々のプロセス
＊共有メモリセグメント
＊セマフォ、メッセージキュー
＊ソケット、ネットワークパケット、ネットワークインタフェース、およびルーティングテーブル照会

本発明の例示的な実施形態によるこのコンテインメント手法をサポートするように変更されたデータ構造体の完全なリストを本明細書に添付した付録７．１に示す。上で説明したように、タグの割り当ては主に継承を通して行われ、ｉｎｉｔ－プロセスはまずコンパートメント０に割り当てられる。プロセスによって作成されるあらゆるカーネルオブジェク

トは、実行中プロセスのカレントラベルを継承する。カーネルの適切なポイントで、アクセス制御チェックが、別のコンパートメントの資源へのアクセスが許可されているコンパートメントを示す規則テーブルを照会する動的ロード可能セキュリティモジュールへのフックの使用を通して行われる。これは、実行中アプリケーションにトランスペアレントに行われる。

【０１５９】

各セキュリティチェックでは、規則テーブルを照会する。上に述べたように、各規則は以下の形態を有する。

【０１６０】

source->destination method m[attr][netdev n]

ただし、ｓｏｕｒｃｅ／ｄｅｓｔｉｎａｔｉｏｎは、以下のうちの１つである。

【０１６１】

ＣＯＭＰＡＲＴＭＥＮＴ（名前の付いたコンパートメント）
ＨＯＳＴ（固定ＩＰｖ４アドレス）
ＮＥＴＷＯＲＫ（ＩＰｖ４サブネット）
ｍ：サポートされるカーネル機構。たとえば、ｔｃｐ、ｕｄｐ、ｍｓｇ（メッセージキュー）、ｓｈｍ（共有メモリ）等
ａｔｔｒ：メソッドｍをさらに修飾する属性
ｎ：妥当な場合には、名前の付いたネットワークインタフェース、たとえばｅｔｈ０
「ＷＥＢ」と名前の付いたコンパートメント中のプロセスが、たとえば、ｓｈｍａｔ／ｓｈｍｄｔ（　）を使用して、「ＣＧＩ」と名前の付いたコンパートメントから共有メモリセグメントへアクセスするのを許可するような規則の例は、以下のようなものである。

【０１６２】

COMPARTMENT:WEB->COMPARTMENT:CGI METHOD shm

特定の暗黙の規則も存在し、コンパートメント内で実行されるある通信を許可する、たとえば、プロセスによる、同じコンパートメントに常駐するプロセスのプロセス識別子の検査を許可することができる。これにより、そうでなければ構成されていないコンパートメント内の機能を必要最小限にすることができる。例外はコンパートメント０であり、これは相対的に非特権であり、より多くの制限が課される。コンパートメント０は通常、カーネルレベルスレッド（スワッパ等）をホストするために使用される。

【０１６３】

コンパートメント間アクセスの実行を明示的に許可する規則がない場合、かかる試みはすべて失敗する。規則の最終的な効果は、別のコンパートメントの資源へのアクセスが明示的に許可されているものを除き、個々のコンパートメントにわたって強制セグメント化を実施することである。

【０１６４】

規則は方向的な性質のものであり、ＴＣＰソケット接続の接続／受け入れの振る舞いに合致する影響を有する。許される入力ＨＴＴＰ接続の指定に使用される以下の形式の規則を考える。

【０１６５】

HOST*->COMPARTMENT X METHOD TCP PORT 80

この規則は、ポート８０上の入力ＴＣＰ接続のみが許可され、出力接続は許可されないことを指定する（図７参照）。規則の方向性により、出力接続の実行を許可することなく、入力接続を正しく確立するために、パケットの逆流を行うことが可能である。

【０１６６】

上に述べた手法にはいくつかの利点がある。たとえば、サポートされる各サブシステムに対して完全なる制御を提供するとともに、サポートされていないもの、たとえば、ハード

ウェア駆動のカード間転送をコンパイルアウトする機能を提供する。さらに、この手法は、ｐｓ、ｎｅｔｓｔａｔ、ｒｏｕｔｅ、ｉｐｃｓ等のユーザ空間コマンドを変更する必要なく、比較的包括的な名前空間区分けを提供する。プロセスが現在入っているコンパートメントに応じて、可視の識別子のリストが規則に何が指定されているかに従って変更される。名前空間の例としては、プロセステーブル、ｖｉａ／ｐｒｏｃ、ＳｙｓＶ　ＩＰＣ資源識別子、アクティブで閉じられたリスニング中のソケット（すべてのドメイン）、およびルーティングテーブルエントリが挙げられる。

【０１６７】

この手法の別の利点は、カーネルおよびその実行中プロセスに関しての同期状態である。スカラータグが各種カーネル資源に添付されることを鑑みて、完全な生存期間にわたる追跡を行う必要がなく、これは、必要なカーネル変数が作成／消費される場所を深く理解する必要があまりないため、パッチを最新に保つという問題を考えた際に大きな利点である。さらに、セキュリティタグの継承は、＃ｉｆｄｅｆｓおよびクローンルーチンを使用することにより明示的に指定する必要がある場合とは異なり、普通のＣ割り当て演算子（＝）またはｍｅｍｃｐｙ（　）を通して自動的に行われるため、行う必要のあるソース変更はより少ない。

【０１６８】

加えて、アクティブ化されるときに、カーネル資源を再帰的に列挙する必要がない。これは、かかる計算がカーネル開始時に行われるためである。さらに、この手法は、行われるソース変更の数が比較的少ないため、比較的高速なパフォーマンス（最適の約１～２％）を提供する。システムの使用意図に応じて、内部ハッシュテーブルは、挿入された規則が各ハッシュバケット内で平均して１レベル深さである（これにより、規則参照ルーチンの振る舞いがＯ（１）のオーダになる）ように構成することができる。

【０１６９】

しかし、多くの利点に関わらず、この手法では、ソース変更をカーネルに対して行う必要があり、また、新しいカーネル改訂が入手可能になったときにパッチをアップデートする必要がある。さらに、構造サイズが異なる可能性があるため、モジュールとして配布されるプロプラエタリデバイスドライバは使用不可である。

【０１７０】

２．動的ロード可能カーネルモジュールを介してのシステムコール置換（Ｖ２）

この手法は、動的ロード可能カーネルモジュールの形態でコンテインメントを実施することを含み、上に概説したソースレベルカーネル変更手法の機能を、カーネルソースを変更する必要なく再現することを意図した手法である。

【０１７１】

この手法では、モジュールは、選択されたシステムコールをｓｙｓ＿ｃａｌｌ＿ｔａｂｌｅ［］アレイを上書きすることによって置換し、また、それ自体をｎｅｔｆｉｌｔｅｒモジュールとして登録して、入力／出力ネットワークパケットをインタセプトする。モジュールは、システム上の各実行中プロセスが要求する資源を反映し、インタセプトされたシステムコールの適切な時点でアップデートされるプロセスＩＤ（ＰＩＤ）駆動内部状態テーブルを保持する。これらテーブルは、所望の実施態様に応じてプロセス単位あるいは資源単位でセキュリティ属性も含むことができる。

【０１７２】

この手法の規則形式およびシンタックスは実質的に、上に概説したソースレベルカーネル変更手法に関して述べたようなものであり、同様に振る舞う。セグメント化は、システムコールレイヤにおける名前空間の区分けを通して行われる。元のシステムコールを介してのカーネル資源へのアクセスは、実際にシステムコールを行う前に行われるセキュリティチェックを条件としている。

【０１７３】

すべてのシステムコールの置換は、この手法におけるシステムコールの取り扱い方の条件的な性質を反映した独特なｐｒｅ／ａｃｔｕａｌ／ｐｏｓｔ形態を有する。

【０１７４】
したがって、この手法には、カーネル変更が必要ないという利点があるが、カーネル内部の知識が要求される。さらに、セキュリティモジュールが一時的にディセーブルされている間にシステムを実行する機能により、バグの分類がより容易になる。
【０１７５】
この手法と併せて考慮すべきいくつかの欠点および／または問題もある。第１に、実行中のプロセスに関して真に同期する状態を保つことは、主に包括的なカーネルイベント通知機構の欠如に起因する様々な理由から困難である。たとえば、ＳＩＧＳＥＧＶ、ＳＩＧＢＵＳ等によりプロセスが異常に終了する状況を捕捉する正式な機構はない。この問題に提案されている１つの方策は、ｄｏ＿ｅｘｉｔ（　）に対してわずかなソースコード変更を行い、かかる場合を捕捉するコールバックを提供することを含む。例示的な一実施形態では、カーネルレベル刈り取り（reaper）スレッドを使用して、グローバルタスクリストを監視し、死んだＰＩＤに対してガーベッジコレクション（ごみ収集）を行うことができる。これにより、不安定な短い期間が導入されるが、これは、ＰＩＤのサイクルが上向きであること、および刈り取りスレッドの単一サイクル内で以前使用されたＰＩＤが再度割り当てられる可能性が比較的低いことによっていくらか相殺される。
【０１７６】
上に述べた子の暴走問題に関して、ｆｏｒｋ／ｖｆｏｒｋ／ｃｌｏｎｅは、おそらくは子が実行するようにスケジューリングされた後まで、子のＰＩＤを戻さない。モジュール実施によりＰＩＤ駆動状態テーブルが作成される場合、これは、子に対する状態エントリが作成される前に、子がシステムコールを呼び出す可能性があることを意味する。子プロセスに添付する必要性があるためにフォークされた子を適宜辿ることができないｓｔｒａｃｅコマンド（上述）にも同じ問題がある。この問題に対して考えられる１つの方策は、必須条件チェックですべてのシステムコールをインタセプトすることであるが、この方策は比較的遅く、状況によっては効果がない。
【０１７７】
考えられる別の方策は比較的複雑であり、本明細書に添付した付録７．２に示される。
【０１７８】
１．ｆｏｒｋ（　）－親のスタック上のリターンアドレスが、スタックをユーザ空間に書き込むことによって実際のｆｏｒｋ（　）システムコールを呼び出す前に変更される。これは、変更されたリターンアドレスを継承する子に引き継がれる。変更されたリターンアドレスは元の値よりも５バイト前のポイントにセットされ、これによりｆｏｒｋ（　）システムコールが子によって最初のアクションとして再び呼び出される。次いで、システムはこれをインタセプトし、必要な状態エントリを作成する。親は、ｆｏｒｋ（　）から戻る直前に、保存されているリターンアドレスを回復させ、通常通り続行する。（５バイトはまさにＩＡ－３２のｆａｒ　ｃａｌｌの形態の命令の長さであることに留意する。他のバリアントは、ＬＤ＿ＰＲＥＬＯＡＤおよび所望の５バイト形態を有するシステムコールラッパを使用してラップすることができる）
２．ｃｌｏｎｅ（　）－スタックのセットアップが異なるため、フォークされた子（上述したように）がクローン化された子を取り扱うには不適切な場合に用いられるメソッド。代わりに提案されている方策は、以下である。
【０１７９】
ａ．ユーザプロセスの代わりにｂｒｋ（　）を呼び出し、小さな２５６バイトの塊のメモリを割り当てる。
【０１８０】
ｂ．用意された実行可能コードの塊を新たに割り当てられたこのメモリにコピーする。このコードは、クローン化された子を通常通り処理する前に、指定されたシステムコールを呼び出すであろう。
【０１８１】
ｃ．ｃｌｏｎｅ（　）への呼び出しにおいて与えられる元のルーチンの代わりに、この新

たに用意されたコードの塊を実行するように、ユーザプロセスのスタックを変更する。
【０１８２】
ｄ．ユーザプロセスによって与えられたルーチンへの元のポインタをクローンに保存する。
【０１８３】
クローン化された子が最初に実行されるとき、ポインタを、実行すると想定されていた元のルーチンに戻すシステムコールを行う、用意されたコードの塊を実行する。子はこのポイントでトラップされ、これに対して状態エントリが作成される。次いで、クローン化された子が通常通り元のルーチンを実行する（本明細書に添付した付録７．４を参照）。
【０１８４】
両方の場合において、子はカーネルモジュールに下がって呼び出すように強制され、そこで子をトラップすることができる。
【０１８５】
考えられる別の方策は、子が作成される都度、コールバックを提供するように、カーネルにおけるｒｅｔ＿ｆｒｏｍ＿ｆｏｒｋ（　）ルーチンを変更することである。代替としては、ｆｏｒｋ／ｖｆｏｒｋ／ｃｌｏｎｅを実施するｄｏ＿ｆｏｒｋ（　）カーネル関数を変更することができる。
【０１８６】
ｃｌｏｓｅ－ｏｎ－ｅｘｅｃ振る舞いの追跡もまた、この実施態様では、各プロセス構造体内のファイルシステム関連構造体を熟知していなければ難しい。
【０１８７】
この手法と併せて考慮すべき別の問題は、カーネル資源の事後列挙が、ブートシーケンスが進むにつれて累進的に難しくなることから、可能な限り早くカーネル資源の監視を開始するために、通常、モジュールがブートシーケンスのかなり初期にロードされることである。この手法においてシステムコール引数の有効性をチェックするプロセスが、元のシステムコールではなくカーネルモジュールにシフトされることにも留意されたい。したがって、元のカーネルは変更されないため、この手法ではオーバヘッドがさらにもたらされる。同様に、実質的に複製された状態情報をカーネルから離れて維持することにより、メモリ使用およびプロセッササイクルに関してオーバヘッドが追加される。
【０１８８】
さらに別の欠点は、コンパートメント毎のルーティングおよびこれに依存する機構、すなわち仮想化ＡＲＰキャッシュ、ならびにルートを使用してバックエンドネットワークアクセスをセグメント化する機能が失われることである。これは、タグ付きデータ構造体なしでルーティングコードが変更されずに実行されるためである。最後に、すべての構成を満足させる単一バイナリモジュールを提供することは、不可能でないとしても非常に困難であると考えられる。構造体内のデータメンバのサイズおよびレイアウトは、その特定のカーネルビルドにおける構成オプションに依存する。たとえば、ｎｅｔｆｉｌｔｅｒのコンパイルを指定すると、あるネットワーキング関連データ構造体のサイズおよびレイアウトが変更される。
【０１８９】
動的ロード可能カーネルモジュールの配備と併せて考慮すべきいくつかの問題がある。特定のカーネルデータ構造体のサイズは、構築時に決定される実際の構成オプションに依存する、すなわちデータメンバの数は、カーネルにおいてコンパイルするように選択された機能が何であるかに応じて可変であるため、モジュールをカーネルに整合させる必要性が極めて重要である。したがって、モジュールは既知のカーネルに対して構築することができ、この場合、ソースおよび構成オプション（構成ファイルに表される）を容易に入手可能であり、あるいはインストール時に構築することができ、この場合、モジュールへのソースはインストール場所まで運ぶ必要がある。
【０１９０】
３．カーネルベースの変更をサポートするハイブリッドシステムコール置換

図面の図８を参照して、上に述べた変更カーネルベースの手法（Ｖ１）とシステムコール置換手法（Ｖ２）の特徴のいくつかを組み合わせたハイブリッドコンテインメントオペレーティングシステムの構築に利用可能なオプションのいくつかを模式的に示す。

【０１９１】

実行中のカーネルに関しての状態維持について、Ｖ１手法は、適切な通知機構が欠如していること、およびガーベッジコレクション（ごみ収集）が必要なことから歩調がわずかにずれたままであるＶ２と比較して、カーネルの実際の動作とはるかに密接に歩調を揃えている。Ｖ１での状態情報は、カーネルに関して厳密に同期し、Ｖ２は非同期である。同期性は、内部状態テーブルが、通常、同期プリミティブの取得によってバウンディングされる同じコードセクション内の実際のカーネル状態の変更に伴ってロックステップでアップデートされるか否かによって決まる。同期の必要性を図面の図９に示し、図中、組み込まれたソースから生じるカーネル状態の変更は、介在レイヤにおける複製状態に反映する必要がある。

【０１９２】

再び図面の図８を参照すると、Ｖ１およびＶ２の手法を併せた相対的な利点は、開発者が、略同期状態を実現するためにカーネルソースを変更したいと望む強度に応じて、Ｖ１手法が代表する同期状態の位置と、Ｖ２手法によって提供される非同期状態の位置との間で可変的に決められる。図８は、Ｖ２手法を変更することで、カーネルソースコード変更という相対的にわずかな犠牲で大きな利点が得られる３つのポイントを示す。

【０１９３】

１．ｄｏ＿ｅｘｉｔ（　）－ｄｏ＿ｅｘｉｔ（　）カーネル関数を５ライン変更すると、コールバックを提供して、プロセスの異常終了によるグローバルタスクリストの変更を捕捉することが可能になる。かかる変更は、プロセス終了がどのように処理されたかについて知る必要はないが、制御パスがどこにあるかを理解する必要がある。

【０１９４】

２．ｆｏｒｋ／ｖｆｏｒｋ／ｃｌｏｎｅ－ｄｏ＿ｆｏｌｋカーネル関数をさらに５ライン変更すると、実行するようにスケジューリング可能になる前に子のＰＩＤを適宜通知することが可能になる。代替は、ｒｅｔ＿ｆｒｏｍ＿ｆｏｒｋ（　）を変更することであるが、これはアーキテクチャ依存である。これらオプションのいずれでも、プロセスセットアップの知識は必要なく、ＰＩＤ作成の性質およびＰＩＤ関連構造体を取り巻くロックを認識している必要があるだけである。

【０１９５】

３．割り込み、ＴＣＰタイマ等－このカテゴリは、ハード／ソフトＩＲＱ、ｔａｓｋｌｅｔ、内部タイマ、またはユーザプロセスにトレース不可能なあらゆる実行コンテキストのいずれかの結果としてカーネルにおいて非同期に実行されるすべての動作を網羅する。例は、閉じられたが、まだ完全に消失していないソケットを維持するために使用されるＴＣＰ時間待ちハッシュバケットである。ハッシュテーブルは、パブリックにエクスポートされず、ハッシュテーブルの変更は、コールバック用に正式なＡＰＩがないため追跡することができない。パケット単位で計算を行う必要がある（これは、Ｖ１手法の主要な利点であり、ここからいくつかの特徴が導出される）場合、このカテゴリのカーネルソースへの変更が必要である。しかし、これら（比較的広範な）変更を実行するためには、サブシステムの内部作業について熟知している必要がある。

【０１９６】

本発明の最も重要な用途の１つは、任意のＣＧＩバイナリの制限実行をサポートし、非ＨＴＴＰ関連処理（たとえば、Ｊａｖａサーブレット）がいずれも別個のコンパートメントに区分けされ、それぞれの動作に必要な最低限の規則をそれぞれ有する安全なウェブサーバプラットフォームの提供である。これは、以下の一般的なシナリオよりも具体的な構成である。

【０１９７】

１．ＤＮＳ、Ｓｅｎｄｍａｉｌ等の様々なサービスをホストする安全なゲートウェイシス

テム。かかるシステムにおけるコンテインメントまたはコンパートメント化を用いて、サービス間の競合の可能性を低減し、また、サービス単位でバックエンドホストの可視性を制御することができる。

【０１９８】

２．中間アプリケーションサーバを含む積層バックエンドに対するフロントエンド（通常ＨＴＴＰ）のクラスタ化。かかるシステムにおけるコンパートメント化は、外部クライアントから直接アクセス可能なコードを可能な限り多く取り除くといった望ましい効果を有する。

【０１９９】

要約すれば、本発明の背後にある基本原理は、あらゆる外部アクセス可能コードのサイズおよび複雑性を最小まで低減することであり、これによって実際のセキュリティ侵害が発生する可能性がある範囲を制限する。可能な限り狭いインタフェースが、可能な限り具体的な規則を使用することにより、かつ／または規則の方向性を利用することにより、個々のコンパートメントにグループ化された各種機能コンポーネントの間に指定される。

【０２００】

これより、図面の図２を参照して、選ばれた手法であるＶ１に基づいて構成されたウェブサーバプラットフォームを示す。上に述べたように、各ウェブサーバは、それぞれのコンパートメントに置かれる。ＭＣＧＡデーモンは、ＣＧＩ実行要求を取り扱い、それぞれのコンパートメントに置かれる。同様に、管理目的のコンパートメントもさらにある。ユーザレベルコマンドラインユーティリティを利用して、規則の追加／削除、およびプロセスラベルのセットによりカーネルを構成する管理ＣＧＩユーティリティも示される。これらユーティリティは、特権デバイスドライバインタフェースを介して動作する。カーネルでは、各サブシステムが、規則および初期にセットされた構成情報にのっとって動作するカスタムセキュリティモジュールへのコールアウトを含む。システムコールを行うユーザプロセスは最終的に、各サブシステムに存在するセキュリティチェックを受け、対応するデータが処理され、適宜タグ付けされる。

【０２０１】

以下の説明は、本発明をどのように使用して、Ｊａｖａサーブレットの取り扱いまたはＪＳＰファイルの扱いを、各自のコンパートメント内でそれぞれ実行中の２つの別個のインスタンスＪａｋａｒｔａ／Ｔｏｍｃａｔに委譲するように構成された、外部を向いたＡｐａｃｈｅウェブサーバを含むセットアップをコンパートメント化することができるかについての説明を意図するものである。デフォルトにより、各コンパートメントは、その他のコンパートメントに干渉しないように、ｃｈｒｏｏｔ化されたファイルシステムを使用する。

【０２０２】

図面の図１０は、１つのコンパートメント（ウェブ）に常駐するＡｐａｃｈｅプロセスを模式的に示す。このコンパートメントは、以下の規則を用いて外部からアクセス可能である。

【０２０３】

HOST*->COMPARTMENT WEB METHOD TCP PORT 80 NETDEV eth0

規則中のＮＥＴＤＥＶコンポーネントの存在により、Ａｐａｃｈｅが使用可能なネットワークインタフェースが指定される。これは、デュアル／マルチホームゲートウェイシステム上の外部インタフェースのみを使用するようにＡｐａｃｈｅを制限する際に有用である。これは、不正侵入されたＡｐａｃｈｅのインスタンスが、内部を向いたネットワークインタフェースを通してバックエンドネットワークに対する攻撃の開始に使用されることの回避を意図する。ウェブコンパートメントでは、以下の形態をとる２つの規則を介して、Ｊａｋａｒｔａ／Ｔｏｍｃａｔの２つの別個のインスタンス（ＴＯＭＣＡＴ１およびＴＯＭＣＡＴ２）との通信が許可されている。

【０２０４】

COMPARTMENT:WEB->COMPARTMENT:TOMCAT1 METHOD TCP PORT 8007

COMPARTMENT:WEB->COMPARTMENT TOMCAT2 METHOD TCP PORT 8008

ＴＯＭＣＡＴ１中のサーブレットは、次の規則を使用して、サーバ１と呼ばれるバックエンドホストにアクセス許可されている。

【０２０５】

COMPARTMENT:TOMCAT1->HOST:SERVER1 METHOD TCP …

しかし、ＴＯＭＣＡＴ２は、いずれのバックエンドホストにもアクセス許可されていない。これは、任意の追加規則がないことに反映されている。カーネルは、ＴＯＭＣＡＴ２からのかかる試行をいずれも拒絶するであろう。これにより、どのサービスがホストされているかに応じてバックエンドネットワークのビューを選択的に変更し、また、バックエンドホストの可視性をコンパートメント単位で制限することが可能になる。

【０２０６】

上記４つの規則だけがこの例示的な構成に必要なものであることはまったく価値がない。任意の他の規則がない場合、ＪａｖａＶＭで実行中のサーブレットは、出力接続を開始することができず、特に、インタフェースｅｔｈ１上の内部バックエンドネットワークに対する攻撃の開始に使用することはできない。加えて、他のコンパートメント（たとえば、共有メモリセグメント、ＵＮＩＸドメインソケット等）から資源にアクセスすることができない場合があり、また、リモートホストが直接到達できない場合がある。この場合、ＡｐａｃｈｅおよびＪａｋａｒｔａ／Ｔｏｍｃａｔの振る舞いに対して、それぞれソースを再コンパイルまたは変更することなく、強制制限が課されている。

【０２０７】

これより、アプリケーション統合の例について、ＯｐｅｎＭａｉｌ１６．０を参照して述べる。Ｌｉｎｕｘ版ＯｐｅｎＭａｉｌ１６．０は、いくつかの未指定フォーマットの大きな１６０Ｍｂ＋アーカイブおよびインストールスクリプトｏｍｉｎｓｔａｌｌからなる。ＯｐｅｎＭａｉｌをインストールするためには、割り当てられた必要最小限度の内部コンパートメントにｃｈｒｏｏｔ化を行うことがまず必要である。

【０２０８】

root@tlinux# chroot/compt/omailin
root@tlinux# ominstall
root@tlinux# ［ＯｐｅｎＭａｉｌが自然にインストールされるのを待つ］
root@tlinux# ［必要であれば、さらなる構成、たとえばｍａｉｌｎｏｄｅセットアップを行う］

ＯｐｅｎＭａｉｌ１６．０は、これもまたインストールする必要があるウェブベースのインタフェースであるため、別の必要最小限度のコンパートメントが割り当てられ（ｏｍａｉｌｏｕｔ）、ＡｐａｃｈｅＨＴＴＰサーバがＨＴＴＰクエリの処理のためにインストールされる。

【０２０９】

root@tlinux# chroot/compt/omailout
root@tlinux# rpm—install〈apache-RPM-filoename〉
root@tlinux# ＯｐｅｎＭａｉｌのインストール命令に必要なＣＧＩ要求を扱うようにＡｐａｃｈｅのｈｔｔｐｄ．ｃｏｎｆを構成］

この時点で、ＡｐａｃｈｅＨＴＴＰサーバがアクセスできるように、ＯｐｅｎＭａｉｌ６．０に付属のＣＧＩバイナリをインストールする必要もある。これは、２つの方法のうちの一方によって行うことができる。

【０２１０】

＊ＯｐｅｎＭａｉｌをｏｍａｉｌｏｕｔに再びインストールし、不必要な部分、たとえば

サーバプロセスを除去する、または
＊ＯｐｅｎＭａｉｌ　ＣＧＩバイナリをｏｍａｉｌｉｎからコピーし、許可およびディレクトリ構造体を保持するように取り計らう。
【０２１１】
いずれの場合でも、ＣＧＩバイナリは通常、Ａｐａｃｈｅウェブサーバのｃｇｉ－ｂｉｎディレクトリに置かれる。ディスクスペースが問題ではない場合には、前者の手法がより強力であり、うまくいく。後者の方法は、厳密にどのバイナリを外部に向けられたｏｍａｉｌｏｕｔコンパートメントに配置するかを確実にする必要がある場合に使用することができる。最後に、両方のコンパートメントを開始することができる。
【０２１２】
root@tlinux#　comp_start omailout omailin
異なる発信元コンパートメント番号を有するＩＰフラグメントを受け取る場合もあり得る。かかる場合、システムは、フラグメントの再アセンブリが、コンパートメント番号の異なるフラグメントを続行するのを許可しない手段を備えることができる。
【０２１３】
他の様々なネットワークプロトコル、たとえばＩＰＸ／ＳＰＸ等へのサポートも含むことができる。
【０２１４】
ｃｈｒｏｏｔ－ｊａｉｌよりも包括的なファイルシステム保護方法を使用し得るものと考えられる。
【０２１５】
図面の図１３を参照して、本発明の例示的な実施形態の動作を模式的に示す。ゲートウェイシステム６００（内部および外部ネットワークの両方に接続された）を示す。ゲートウェイシステム６００は、複数のタイプのサービス：サービス０、サービス１、・・・、サービスＮをホストしており、各サービスは指定されたあるバックエンドホスト、すなわちホスト０、ホスト１、・・・、ホストＸ、ホストＮに接続され、その機能、たとえばバックエンドデータベースからの記録検索を行う。多くのバックエンドホストが、随時内部ネットワークに存在してよい（すべてのバックエンドホストが同じサービスセットによりアクセス可能であるわけではないよう意図される）。これらサーバプロセスが不正侵入されても、不正侵入されたサーバプロセスを使用して、サービスによる使用を当初意図されていない他のバックエンドホストをプロービングすることはできないはずである。本発明は、同じネットワーク上のホストの可視性を制限することにより、攻撃者が与え得る損害を制限するよう意図される。
【０２１６】
図１３では、サービス０およびサービス１のみが、ネットワークインタフェースｅｔｈ０を通してネットワークサブネット１にアクセス許可されている。したがって、ホスト０／ホスト１がサブネット１であるため、ホスト０／ホスト１にアクセスする試みは成功するが、ｅｔｈ１を介してサブネット２にアクセスする試みは失敗する。さらに、サービスＮは、ｅｔｈ１上のホストＸのみへのアクセスが許可されている。このため、ホストＮがホストＸと同じサブネット上にあってもサービスＮによるホストＮへのアクセス試行はいずれも失敗し、また、サービスＮによるサブネット１上のあらゆるホストへのアクセス試行は失敗する。
【０２１７】
制限は、（規則またはルーティングテーブルにより）サブネットまたは特定のホストによって指定することができ、同様に、特定のサブネットによっても限定することができる。
【０２１８】
図面の図１４を参照して、本発明者らによる第１の同時係属中の国際出願の発明の第４の態様の例示的な実施形態によるオペレーティングシステムの動作を模式的に示す。この発明のこの態様の例示的な実施形態の主な好ましい特徴は、以下である。
【０２１９】

１．ｒｏｏｔへの移行が可能なエリアにおけるオペレーティングシステムのソースコードに対する変更。フックがこれらポイントに追加され、フックにより、実行時に、遷移の実行を許可あるいは拒絶する関数をコールアウトする。

【０２２０】

２．各実行中プロセスにタグをマークするという、オペレーティングシステムのソースコードに対する変更。上に述べたように、生成されるプロセスは、それぞれのタグをそれぞれの親プロセスから継承する。特別な特権プログラムが、それ自体のタグとは異なるタグを有する外部プログラムを起動することができる（異なるタグを有するプロセスでシステムを埋めるための手段）

３．構成ユーティリティが、特定のタグに関連するいずれのプロセスに「封印」とマークすべきかを実行時にオペレーティングシステムに対して指定することができるようにする機構

４．上記構成ユーティリティに渡すべきデータを記述する構成ファイル

したがって、本発明は、ファイルまたはプログラムにアクセスするときにアクセスされないパスベースの規則仕様を使用して、機能が主にカーネルレベルで提供される高信頼性オペレーティングシステム、特にＬｉｎｕｘベースの高信頼性オペレーティングシステムを提供する。これは、ディスクに格納されているプログラムまたはファイルではなく実行中のプロセスにおいてあらゆる管理特権を推論することによって実現される。かかる特権は、アクティブ化されると管理タグまたはラベルの継承により付与されるため、組み込まれたセキュリティ属性のためにストリームまたはパケットを後に復号化する必要はない。これは、ストリームまたはパケットがそれぞれのセキュリティ属性に従って異なるパスに沿って再ルーティングされないためである。

【０２２１】

Ｌｉｎｕｘの機能には、ユーザ空間における高信頼性アプリケーションを必要とすることなくアクセス可能であり、実行中プログラムのセキュリティレベルのアップグレード、ダウングレード、もしくは変更は必要ない。

【０２２２】

本発明の実施形態について例としてのみ上述したが、併記の特許請求の範囲に規定される本発明の範囲から逸脱することなく、変更および変形を上記実施形態に行うことができることが当業者には明白であろう。

【図面の簡単な説明】

【０２２３】

【図１】コンテインメントプロパティを有するオペレーティングシステム上でマルチサービスホストとして機能するための例示的なアーキテクチャの模式図である。

【図２】本発明の例示的な実施形態による高信頼性Ｌｉｎｕｘホストオペレーティングシステムのアーキテクチャの模式図である。

【図３】図２に示すオペレーティングシステムにおいて使用される例示的な変更データ型を示す。

【図４】ＬｉｎｕｘＩＰネットワーキングにおける主なネットワーキングデータ型を示す。

【図５】ＩＰネットワーキングのｓｔｒｕｃｔ　ｃｓｅｃｉｎｆｏデータメンバの伝搬を示す。

【図６】コンテインメントをＬｉｎｕｘカーネルに構築する例示的な３つの手法を模式的に示す。

【図７】規則：ＨＯＳＴ＊－＞ＣＯＭＰＡＲＴＭＥＮＴｘＭＥＴＨＯＤ　ＴＣＰ　ＰＯＲＴ８０の影響を模式的に示す。

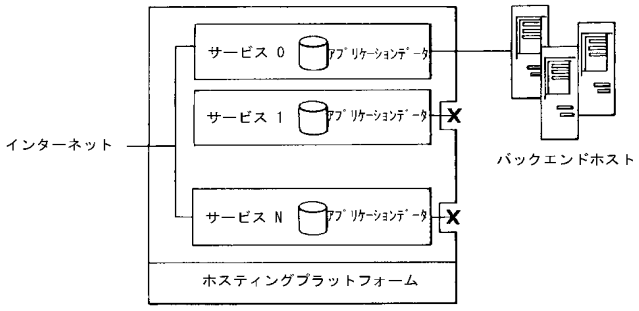
【図８】ハイブリッドコンテインメントプロトタイプオペレーティングシステムの構築に利用可能な多様なオプションを模式的に示す。

【図９】複製したカーネル状態を同期して更新することの望ましさを模式的に示す。

【図１０】ＡｐａｃｈｅおよびＴｏｍｃａｔ　Ｊａｖａ　Ｖｍの例示的な構成を模

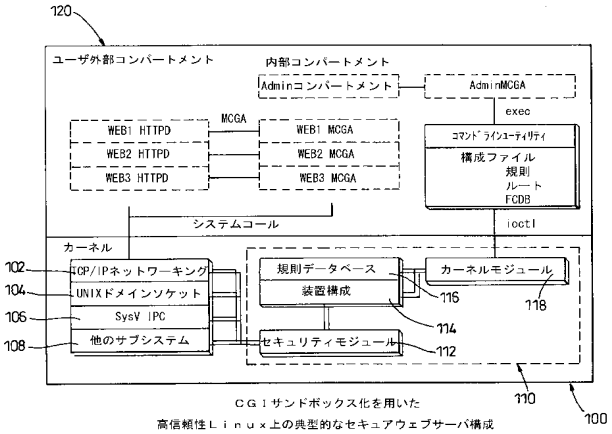
式的に示す。
【図１１】図２に示す高信頼性Ｌｉｎｕｘにおけるｃｈｒｏｏｔ化された層状の環境を模式的に示す。
【図１２】カーネル強制コンパートメントアクセス制御規則の効率的な参照プロセスを模式的に示す。
【図１３】本発明の一態様による高信頼性ゲートウェイシステムの例示的な実施形態を模式的に示す。
【図１４】本発明の一態様の例示的な実施形態によるオペレーティングシステムの動作を模式的に示す。
【図１５】従来技術によるオペレーティングシステムの例示的な実施形態を模式的に示す。

【図１】



コンテインメントプロパティを有する
オペレーティングシステム上で
マルチサービスホストとして機能するアーキテクチャ例

【図２】



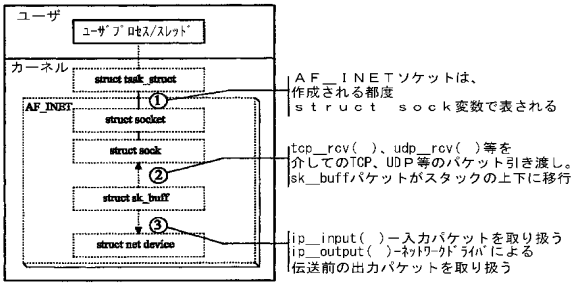
ＣＧＩサンドボックス化を用いた
高信頼性Ｌｉｎｕｘ上の典型的なセキュアウェブサーバ構成

【図３】

```
struct csec info {
        unsigned long sl ;
);
struct sock {
...
#ifdef CASPER
        struct csecinfo csi : /*    コンパートメント番号を含む    */
#endif /* CASPER */
);
```

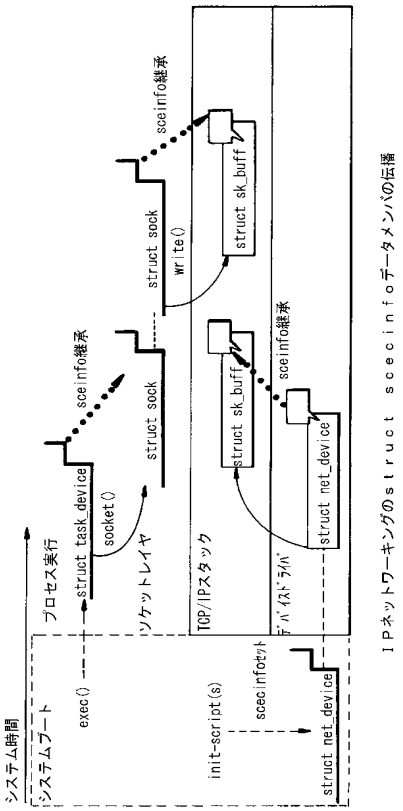
変更されたデータ型の例

【図４】

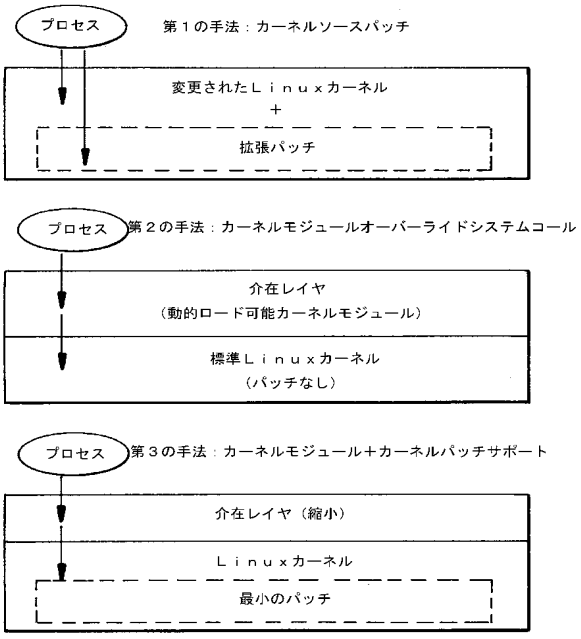


ＬｉｎｕｘＩＰネットワーキングにおける主なネットワーキングデータ型

【図５】

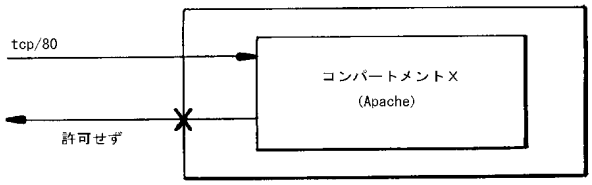


【図６】



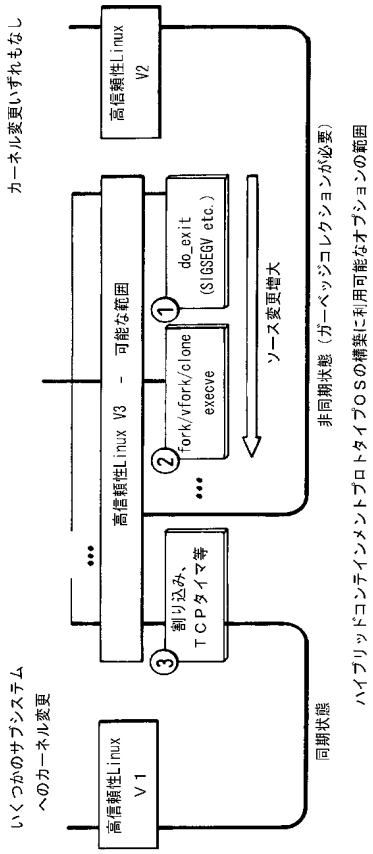
Ｌｉｎｕｘカーネルにコンテインメントを組み込む３つの手法

【図７】

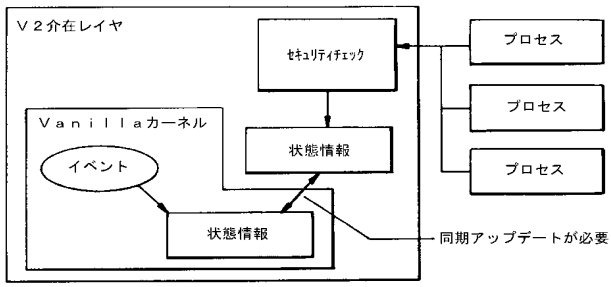


入力ＴＣＰ接続のみ許可

【図８】

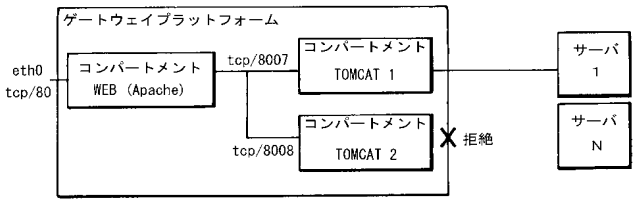


【図９】



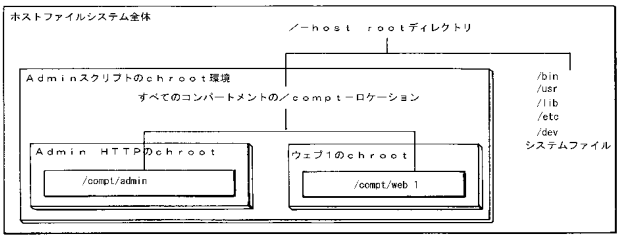
複製されたカーネル状態を同時にアップデートする必要

【図１０】



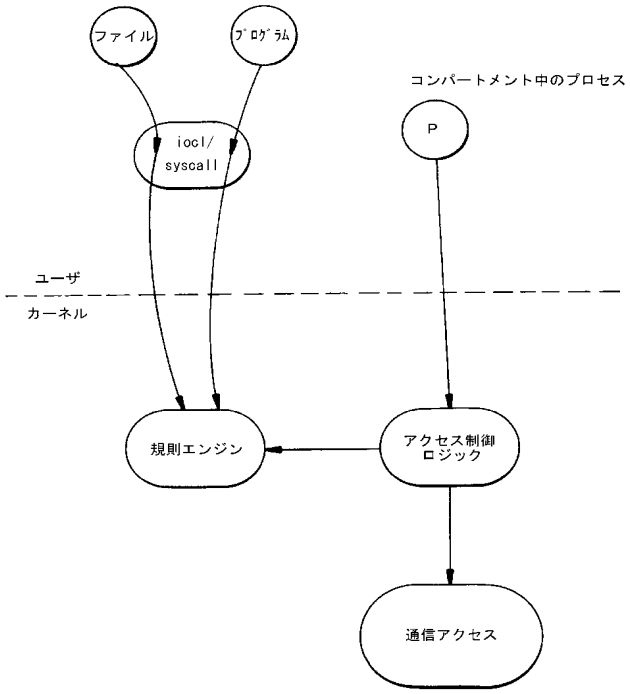
ＡｐａｃｈｅおよびＴｏｍｃａｔ　Ｊａｖａ　ＶＭの構成

【図１１】

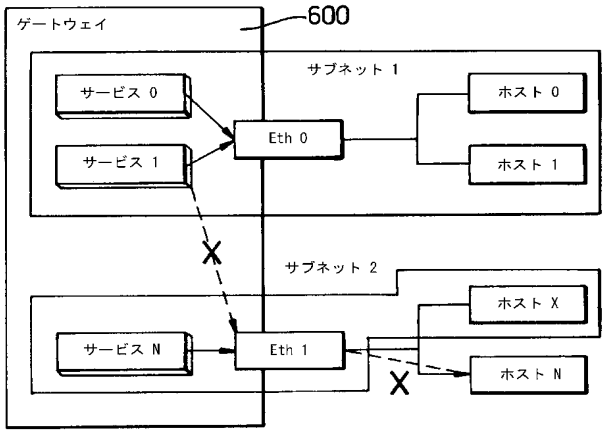


高信頼性Ｌｉｎｕｘにおける層状ｃｈｒｏｏｔ化環境

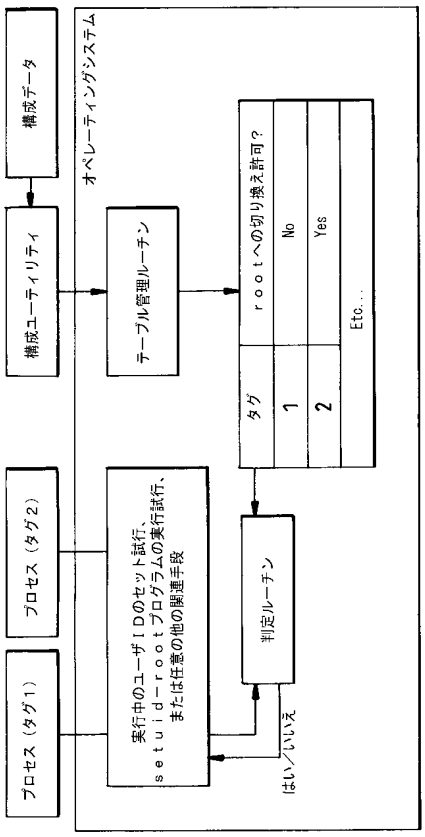
【図１２】



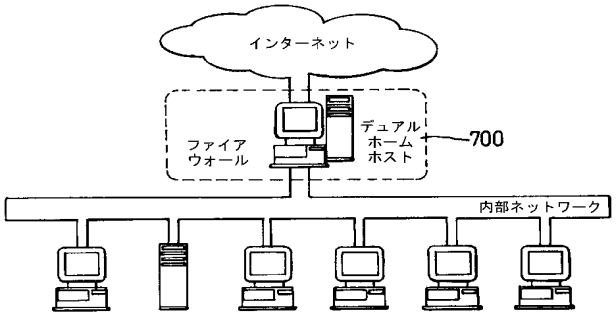
【図１３】


ゲートウェイ
600
サブネット１
サービス０
サービス１
Eth 0
ホスト０
ホスト１
サブネット２
サービスＮ
Eth 1
ホストＸ
ホストＮ

【図１４】


構成データ
構成ユーティリティ
オペレーティングシステム
テーブル管理ルーチン
rootへの切り換え許可?
タグ
1
2
No
Yes
Etc..
プロセス（タグ２）
プロセス（タグ１）
実行中のユーザＩＤのセット試行、
setuid-rootプログラムの実行試行、
または任意の他の関連手段
判定ルーチン
はい／いいえ

【図１５】


インターネット
ファイア
ウォール
デュアル
ホーム
ホスト
700
内部ネットワーク

【国際公開パンフレット】

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
8 August 2002 (08.08.2002)

PCT

(10) International Publication Number
WO 02/061552 A1

(51) International Patent Classification⁷: G06F 1/00

(21) International Application Number: PCT/GB02/00385

(22) International Filing Date: 29 January 2002 (29.01.2002)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
0102516.2 31 January 2001 (31.01.2001) GB

(71) Applicant (for all designated States except US): HEWLETT-PACKARD COMPANY [US/US]; 3000 Hanover Street, Palo Alto, CA 94304 (US).

(72) Inventors; and
(75) Inventors/Applicants (for US only): CHOO, Tse, Huong [MY/GB]; 46 The Culvert, Bradley Stoke, Bristol BS32 8AB (GB). DALTON, Christopher, I. [GB/GB]; 19 Buckingham Road, Redland, Bristol BS6 6TJ (GB). NORMAN, Andrew, Patrick [GB/GE]; 254 Juniper Way, Bradley Stoke, Bristol BS32 0DR (GB).

(74) Agent: LAWRENCE, Richard, Anthony; Hewlett Packard Limited, Intellectual Property Section, Filton Road, Stoke Gifford, Bristol BS34 8QZ (GB).

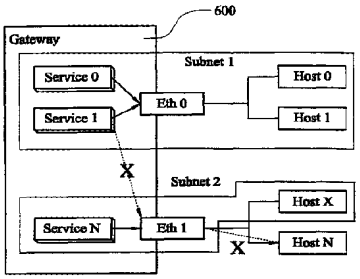
(81) Designated States (national): JP, US

(84) Designated States (regional): European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR).

Published:
— with international search report

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: TRUSTED GATEWAY SYSTEM



(57) Abstract: An operating system comprising a kernel (100) incorporating mandatory access controls as a means to counter the effects posed by application compromise. The operating system uses a technique known as "containment" to at least limit the scope of damage when security breaches occur. In a preferred embodiment, each application supported by the operating system, is assigned a tag or label, each tag or label being indicative of a logically protected computing environment or "compartment", and applications having the same tag or label belonging to the same compartment. By default, only applications running in the same compartment can communicate with each other. Access control rules define very narrow tightly-controlled communications paths between compartments.

WO 02/061552 A1

TRUSTED GATEWAY SYSTEM

Field of the Invention

This invention relates to a trusted operating system and, in particular, to an operating system having enhanced protection against application compromise and the exploitation of compromised applications.

In recent years, an increasing number of services are being offered electronically over the Internet. Such services, particularly those which are successful and therefore lucrative, become targets for potential attackers, and it is known that a large number of Internet security breaches occur as a result of compromise of the applications forming the electronic services.

Background to the Invention

The applications that form electronic services are in general sophisticated and contain many lines of code which will often have one or more bugs in it, thereby making the application more vulnerable to attack. When an electronic service is offered on the Internet, it is exposed to a large population of potential attackers capable of probing the service for vulnerabilities and, as a result of such bugs, there have been known to be security violations.

Once an application has been compromised (for example, by a buffer overflow attack), it can be exploited in several different ways by an attacker to breach the security of the system.

Increasingly, single machines are being used to host multiple services concurrently (e.g. ISP, ASP, xSP service provision), and it is therefore becoming increasingly important that not only is the security of the host platform protected from application compromise attacks, but also that the applications are adequately protected from each other in the event of an attack.

One of the most effective ways of protecting against application compromise at the operating system level is by means of kernel enforced controls, because the controls implemented in the kernel cannot be overridden or subverted from user space by any application or user. In

known systems, the controls apply to all applications irrespective of the individual application code quality.

There are two basic requirements at the system level in order to adequately protect against application compromise and its effects. Firstly, the application should be protected against attack to the greatest extent possible, exposed interfaces to the application should be as narrow as possible and access to such interfaces should be well controlled. Secondly, the amount of damage which a compromised application can do to the system should be limited to the greatest possible extent.

In a known system, the above two requirements are achieved by the abstract property of "containment". An application is contained if it has strict controls placed on which resources it can access and what type of access it has, even when the application has been compromised. Containment also protects an application from external attack and interference. Thus, the containment property has the potential to at least mitigate many of the potential exploitative actions of an attacker.

The most common attacks following the compromise of an application can be roughly categorized as one of four types, as follows (although the consequences of a particular attack may be a combination of any or all of these):

1. **Misuse of privilege to gain direct access to protected system resources.** If an application is running with special privileges (e.g. an application running as root on a standard Unix operating system), then an attacker can attempt to use that privilege in unintended ways. For example, the attacker could use that privilege to gain access to protected operating resources or interfere with other applications running on the same machine.

2. **Subversion of application enforced access controls.** This type of attack gains access to legitimate resources (i.e.

resources that are intended to be exposed by the application) but in an unauthorized manner. For example, a web server which enforces access control on its content before it serves it, is one application susceptible to this type of attack. Since the web server has uncontrolled direct access to the content, then so does an attacker who gains control of the web server.

3. **Supply of false security decision making information.** This type of attack is usually an indirect attack in which the compromised application is usually a support service (such as an authorization service) as opposed to the main service. The compromised security service can then be used to supply false or forged information, thereby enabling an attacker to gain access to the main service. Thus, this is another way in which an attacker can gain unauthorized access to resources legitimately exposed by the application.

Illegitimate use of unprotected system resources. An attacker gains access to local resources of the machine which are not protected but nevertheless would not normally be exposed by the application. Typically, such local resources would then be used to launch further attacks. For example, an attacker may gain shell access to the hosting system and, from there, staged attacks could then be launched on other applications on the machine or across the network.

With containment, misuse of privilege to gain direct access to protected system resources has much less serious consequences than without containment, because even if an attacker makes use of an application privilege, the resources that can be accessed are bounded by what has been made available in the application's container. Similarly, in the case of unprotected resources, using containment, access to the network from an application can be blocked or at least very tightly controlled. With regard to the supply of false security decision making information, containment mitigates the potential damage caused by ensuring that the only

access to support services is from legitimate clients, i.e. the application services, thereby limiting the exposure of applications to attack.

Mitigation or prevention of the second type of attack, i.e. subversion of application enforced access controls, is usually achieved at the application design, or at least configuration level. However, using containment, it can be arranged that access to protected resources from a large untrusted application (such as a web server) must go through a smaller, more trustworthy application.

Thus, the use of containment in an operating system effectively increases the security of the applications and limits any damage which may be caused by an attacker in the event that an application is compromised. Referring to Figure 1 of the drawings, there is illustrated an exemplary architecture for multi-service hosting on an operating system with the containment property. Containment is used in the illustrated example to ensure that applications are kept separated from each other and critical system resources. An application cannot interfere with the processing of another application or obtain access to its (possibly sensitive) data. Containment is used to ensure that only the interfaces (input and output) that a particular application needs to function are exposed by the operating system, thereby limiting the scope for attack on a particular application and also the amount of damage that can be done should the application be compromised. Thus, containment helps to preserve the overall integrity of the hosting platform.

Kernel enforced containment mechanisms in operating systems have been available for several years, typically in operating systems designed for handling and processing classified (military) information. Such operating systems are often called 'Trusted Operating Systems'.

The containment property is usually achieved through a combination of Mandatory Access controls (MAC), and Privileges. MAC protection schemes enforce a particular policy of access control to the system resources such as files, processes and network connections. This policy is enforced by the kernel and cannot be overridden by a user or compromised application.

Despite offering the attractive property of containment, trusted operating systems have not been widely used outside of the classified information processing systems for two main reasons. Firstly, previous attempts at adding trusted operating system features to conventional operating systems have usually resulted in the underlying operating system personalities being lost, in the sense that they no longer support standard applications or management tools, and they can no longer be used or managed in standard ways. As such, they are much more complicated than their standard counterparts. Secondly, previous trusted operating systems have typically operated a form of containment which is more akin to isolation, i.e. too strong, and as such has been found to be limited in scope in terms of its ability to usefully and effectively secure [existing] applications without substantial and often expensive integration efforts.

Our first co-pending International Application defines an arrangement which seeks to overcome the problems outlined above and which provides a trusted operating system having a containment property which can be usefully used to effectively secure a large number of existing applications without application modification.

The first aspect of the invention of our first co-pending International Application provides an operating system for supporting a plurality of applications, wherein at least some of said applications are provided with a label or tag, each label or tag being indicative of a logically protected computing environment or "compartment", each application having the same label or tag belonging to the same compartment, the operating system further comprising means for defining one or more communication paths between said compartments, and means for preventing communication between compartments where a communication path there between is not defined.

The second aspect of the invention of our first co-pending International Application provides an operating system for supporting a plurality of applications, the operating system further comprising a plurality of access control rules, which may beneficially be added from user space and enforced by means provided in the kernel of the operating system, the access control rules defining the only communication interfaces between selected applications (whether local to or remote from said operating system).

This, in the first and second aspects of the invention of our first co-pending International Application, the property of containment is provided by mandatory protection of processes, files and network resources, with the principal concept being based on the *compartment*, which is a semi-isolated portion of the system. Services and applications on the system are run within separate compartments. Beneficially, within each compartment is a restricted subset of the host file system, and communication interfaces into and out of each compartment are well-defined, narrow and tightly controlled. Applications within each compartment only have direct access to the resources in that compartment, namely the restricted file system and other applications within that compartment. Access to other resources, whether local or remote, is provided only via the well-controlled communication interfaces.

Simple mandatory access controls and application or process labeling are beneficially used to realize the concept of a compartment. In a preferred embodiment, each process (or thread) is given a label, and processes having the same labels belong to the same compartment. The system preferably further comprises means for performing mandatory security checks to ensure that processes from one compartment cannot interfere with processes from another compartment. The access controls can be made very simple, because labels either match or they do not.

In a preferred embodiment of the present invention, filesystem protection is also mandatory. Unlike traditional trusted operating systems, the preferred embodiment of the first aspect of the invention of our first co-pending International Application does not use labels to directly control access to the filesystem. Instead, the file systems of the first and second aspects of the invention of our first co-pending International Application are preferably, at least partly, divided into sections, each section being a non-overlapping restricted subset (i.e. a chroot) of the main filesystem and associated with a respective compartment. Applications running in each compartment only have access to the associated section of the filesystem. The operating system of the first and/or second aspects of the invention of our first co-pending International Application is preferably provided with means for preventing a process from transitioning to root from within its compartment as described below with reference to the fourth aspect of the invention of our first co-pending International Application, such that the chroot cannot be escaped. The system may also include means for making selected files within a chroot

immutable.

The flexible but controlled communication paths between compartments and network resources are provided through narrow, tightly-controlled communication interfaces which are preferably governed by one or more rules which may be defined and added from user space by a security administrator or the like, preferably on a per-compartment basis. Such communication rules eliminate the need for trusted proxies to allow communication between compartments and/or network resources.

The containment properties provided by the first and/or second aspects of the invention of our first co-pending International Application may be achieved by kernel level enforcement means, user-level enforcement means, or a combination of the two. In a preferred embodiment of the first and/or second aspects of the invention of our first co-pending International Application, the rules used to specify the allowed access between one compartment and other compartments or hosts, are enforced by means in the kernel of the operating system, thereby eliminating the need for user space interposition (such as is needed for existing proxy solutions). Kernel enforced compartment access control rules allow controlled and flexible communication paths between compartments in the compartmentalized operating system of the first aspect of the invention of our first co-pending International Application without requiring application modification.

The rules are beneficially in the form:

source -> destination method m[attr] [netdev n]

where:

source/destination is one of:

COMPARTMENT (a named compartment)

HOST (possibly a fixed Ipv4 address)

NETWORK (possibly an Ipv4 subnet)

m: supported kernel mechanism, e.g. tcp (transmission control protocol), udp (user-datagram protocol), msg (message queues), shm (shared-

memory), etc.

attr: attributes further qualifying the method m

n: a named network interface if applicable, e.g. eth0

Wildcards can also be used in specifying a rule. The following example rule allows all hosts to access the web server compartment using TCP on port 80 only:

HOST* -> COMPARTMENT web METHOD tcp PORT 80

The following example rule is very similar, but restricts access to the web server compartment to hosts that have a route to the eth0 network interface on an exemplary embodiment of the system:

HOST* -> COMPARTMENT web METHOD tcp PORT 80 NETDEV eth0

Means are preferably provided for adding, deleting and/or listing the access control rules defined for the operating system, beneficially by an authorized system administrator. Means may also be provided for adding reverse TCP rules to enable two-way communication to take place between selected compartments and/or resources.

The rules are beneficially stored in a kernel-level database, and preferably added from user space. The kernel-level database is beneficially made up of two hash tables, one of the tables being keyed on the rule source address details and the other being keyed on the rule destination address details. Before a system call/ISR (Interrupt Service Routine) is permitted to proceed, the system is arranged to check the database to determine whether or not the rules define the appropriate communication path. The preferred structure of the kernel-level database enables efficient lookup of kernel enforced compartment access control rules because when the security check takes place, the system knows whether the required rule should match the source address details or the destination address details, and can therefore select the appropriate hash table, allowing a O(1) rate of rule lookup. If the necessary rule defining the required communication path is not found, the system call will fail.

The third aspect of the invention of our first co-pending International Application provides an operating system for supporting a plurality of applications, said operating system comprising a database in which is stored a plurality of rules defining permitted communication paths (i.e. source and destination) between said applications, said rules being stored in the form of at least two encoded tables, the first table being keyed on the rule source details and the second table being keyed on the rule destination details, the system further comprising means, in response to a system call, for checking at least one of said tables for the presence of a rule defining the required communication path and for permitting said system call to proceed only in the event that said required communication path is defined.

Said encoded tables preferably include at least one hash table.

Often, on gateway-type systems (i.e. hosts with dual-interfaces connected to both internal and external networks), it is desirable to a) constrain the running server-processes to use only a subset of the available network interfaces, b)explicitly specify which remote-hosts are accessible and which are not, and c) have such restrictions apply on a per-process/service basis on the same gateway system.

A gateway system may be physically attached to several internal sub-networks, so it is essential that a system-administrator classifies which server-processes may be allowed to access which network-interface so that if a server-process is compromised from a remote source, it cannot be used to launch subsequent attacks on potentially vulnerable back-end hosts via another network-interface.

Traditionally, fire walls have been used to restrict access between hosts on a per-IP-address and/or IP-port level. However, such fire walls are not fine-grained enough of gateway systems hosting multiple services, primarily because they cannot distinguish between different server processes. In addition, in order to specify different sets of restrictions, separate gateway systems with separate sets of firewall rules are required.

Summary of the Invention

We have now devised an arrangement which seeks to overcome the problems outlined above.

Thus, in accordance with the present invention, there is provided a gateway system having a dual interface connected to both internal and external networks for hosting a plurality of services running processes and/or threads, the system comprising means for providing at least some of said running processes and/or threads with a tag or label indicative of a compartment, processes/threads having the same tag or label belonging to the same compartment, the system further comprising means for defining specific communication paths and/or permitted interface connections between said compartments and local and/or remote hosts or networks, and means for permitting communication between a compartment and a host or network only in the event that a communication path or interface connection there between is defined.

Thus, in the present invention, access control checks are placed, preferably in the kernel/operating system of the gateway system. Such access control checks preferably consult a rule-table which specifies which classes of processes are allowed to access which subnets/hosts. Restrictions can be specified on a per-service (or per-process/thread) level. This means that the view of the back-end network is variable on a single gateway host. Thus, for example, if the gateway were to host two types of services each requiring access to two different back-end hosts, a firewall according to the prior art would have to specify that the gateway host could access both of these back-end hosts, whereas with the present invention, it is possible to specify permitted communication paths at a finer level, i.e. which services are permitted to access which hosts. This increases security somewhat because it greatly reduces the risk of a service accessing a host which it was not originally intended to access.

In a preferred embodiment of the present invention, the access-control checks are implemented in the kernel/operating system of the gateway system, such that they cannot be bypassed by user-space processes.

Thus in a first exemplary embodiment of the present invention, the kernel of the gateway system is provided with means for attaching a tag or label to each running process/thread, the tags/labels indicating notionally which compartment a process belongs to. Such tags may be inherited from a parent process which forks a child. Thus, a service comprising a group of forked children cooperating to share the workload, such as a group of slave Web-server processes, would possess the same tags and be placed in the same 'compartment'. The system

administrator may specify rules, for example in the form:

Compartment X -> Host Y [using Network Interface Z] or
Compartment X -> Subnet Y [using Network Interface Z]

which allow processes in a named compartment X to access either a host or a subnet Y, optionally restricted by using only the network-interface named Z. In a preferred embodiment, such rules are stored in a secure configuration file on the gateway system and loaded into the kernel/operating system at system startup so that the services which are then started can operate. When services are started, their start-up sequence would specify which compartment they would initially be placed in. In this embodiment, the rules are consulted each time a packet is to be sent from or delivered to Compartment X by placing extra security checks, preferably in the kernel's protocol stack.

In a second exemplary embodiment of the present invention, a separate routing-table per-compartment is provided. As in the first embodiment described above, each process possesses a tag or label inherited from its parent. Certain named processes start with a designated tag configured by a system administrator. Instead of specifying rules, as described above with reference to the first exemplary embodiment, a set of configuration files is provided (one for each compartment) which the configure the respective compartment's routing-table by inserting the desired routine-table entries. Because the gateway system could contain an un-named number of compartments, each compartment's routing-table is preferably empty by default (i.e. no entries).

The use of routing-tables instead of explicit rules can be achieved because the lack of a matching route is taken to mean that the remote host which is being attempted to be reached is reported to be unreachable. Routes which do match signify acceptance of the attempt to access that remote host. As with the rules in the first exemplary embodiment described above, routing-entries can be specified on a per-host (IP-address) or a per-subnet basis. All that is required is to specify such routing-entries on a per-compartment basis in order to achieve the same functionality as in the first exemplary embodiment.

As explained above, attacks against running server-processes/daemons (e.g. buffer-overflow, stack-smashing) can lead to a situation where a remote attacker illegally acquires root/administrator-equivalent access on the system hosting the server processes. Having gained administrator access on such a system, the attacker is then free to launch other security breaches, such as reading sensitive configuration/password files, private databases, private keys, etc. which may be present on the compromised system.

Such attacks may be possible if :

a) the server-process runs as administrator and is broken into at run-time due to a software-bug internally;

b) the server-process is initially started as administrator, but was programmed to drop administrator privileges for the duration of most of its operation with the selective ability to regain administrator privileges prior to performing some privileged operation. In such cases, the server-process retains the ability to transition back to root (for some specific purpose) but an attacker, once they have gained control of the process, can do so outside of the original intended purpose;

c) the server-process is initially started as an unprivileged user, but acquires administrator access by subverting the original server-process first and then using that as a means to subvert an external setuid-root program which may be vulnerable in the ways described above.

In accordance with the prior art, one immediate solution to these problems is to plug/fix the specific buffer-overflow bug that initially allowed the attack to occur. The obvious disadvantage to this is, of course, that it is purely reactionary and does not preclude further buffer-overflow bugs from being discovered and exploited in future. Another solution proposed by the prior art, is to arrange for existing functionality in an operating system, e.g. UNIX, to drop all root-equivalent access with the intention of never transitioning back to it. Whilst this prevents the running process from dropping back to root unexpectedly, it does not prevent the program from operating an external setuid-root program that has been, for example, carelessly left lying around and which is vulnerable to being broken if fed some invalid input. If this were to occur, the compromised process running as an unprivileged user could execute the setuid-root program feeding it input that would then cause it to come under

the control of the attacker.

We have devised an arrangement which seeks to overcome the problems outlined above. Thus, the fourth aspect of the invention of our first co-pending International Application provides an operating system for supporting a plurality of applications, the operating system comprising means for providing at least some of said applications with a tag or label, said tags or labels being indicative of whether or not an application is permitted to transition to root in response to a request, means for identifying such a request, determining from its tag or label whether or not an application is permitted to transition to root and permitting or denying said transition accordingly.

In a preferred embodiment, at least one of said tags or labels indicates that an application to which it as attached or with which it is associated is "sealed" therefore immutable.

Thus, the fourth aspect of the invention of our first co-pending International Application introduces a way to stop selected server processes from making the transition to the administrator-equivalent state by marking the processes "sealed" against such state transitions. Whenever those processes attempt to make such a transition, either by invoking a system-routine specifically for such purposes, or by executing an external program marked as 'setuid-root' (i.e. programs which have been previously tagged by the system administrator as having the ability to execute as the administrator regardless of who invoked it), or by any other means, then the operating system will disallow the system-call or the attempt to execute such a marked program.

Advantages provided by the operating system according to the fourth aspect of the invention of our first co-pending International Application include the fact that restriction against root-equivalent access is unconditional and remains in force regardless of how many undiscovered software bugs remain to be exploited in the server-process to be run. If a new exploitable bug is discovered, the restriction remains in place as it did previously with other bugs, regardless of the nature of the new bug. Obviously, this would not be possible in the case where bugs are required to be fixed as they are discovered. Further, the arrangement of fourth aspect of the invention of our first co-pending International Application fixes the external setuid-root

problem where an attacker attempts to subvert an external program that has the capability to run as root instead of the original process. In the arrangement of the fourth aspect of the invention of our first co-pending International Application, any such attempts are tracked in the operating system and the arrangement can be configured to deny the attempt by a marked process from executing such a setuid-root program. In addition, no changes to the original source code of the protected process are required, arbitrary binaries can be run with the assurance that they will not drop back to root.

Trusted Operating Systems typically perform labeling of individual network adapters in order to help determine the required sensitivity label to be assigned to an incoming network packet. Sometimes, other software systems, such as fire walls, perform interface labeling (or colouring as it is sometimes called) to determine which interfaces are to be marked potentially "hostile" or non-hostile. This corresponds to the view of a corporate network as being trusted/secure internally and untrusted/insecure for external Internet links (see Figure 15 of the drawings).

For network adapters (NICs) that remain static during the operation of a computer system, the labeling can be performed during system startup. However, there are classes of NIC which can be dynamically activated on a system, such as "soft" adapters for handling PPP links or any other network-device abstraction (e.g. VLANs, VPNs). Examples of such dynamic adapters include:

* PPP links, e.g. modem connection to an ISP. Typically, a soft adapter is created representing the PPP connection to the ISP.
* Virtual LANs (VLANs) - servers can host software-services operating in a private virtual network using VLANs. Such VLANs can be set up dynamically (on demand, say) so the server hosting such services has to be able to correctly label these interfaces if using a Trusted Operating System or a derivative.

The largely static nature of the configuration shown in Figure 15 of the drawings means that there is little need to handle a new adapter. If a system-administrator wishes to add a new adapter to the dual-homed host 700, he/she would typically bring down the system, physically add the adapter and configure the system to recognize the new adapter properly. However,

this process is not suitable in the case where the system which requires interface labeling has the kind of dynamic interfaces mentioned above.

If no label is applied to the adapter, incoming packets on the adapter would not be assigned correct labels which might violate the security of the system in question. Further, outgoing packets (which presumably have a label correctly assigned to them) cannot be matched correctly against the adapter on which the packet is to be transmitted, therefore violating the security of the system in question.

We have devised an arrangement which seeks to overcome the problems outlined above. Thus, our second co-pending International Application provides an operating system comprising means for dynamically assigning a label to a newly-installed adapter substantially upon activation thereof, the label depending upon the attributes of said adapter, and means for removing said label when said adapter is de-activated.

Thus, when a newly-installed adapter in the operating system is first activated, a label is reliably assigned thereto prior to reception of incoming packets, thereby ensuring that no unlabeled packets are created and passed on to the network protocol stack. Because dynamic adapters are catered for in the operating system of the invention of our second co-pending International Application, new areas of functionality for such labeled systems are opened up, e.g. as a router, mobile device. Further, the label assigned to the adapter can be a function of the run-time properties of the newly-activated adapter. For example, it may be desirable to distinguish between different PPP connections to various ISP's. This cannot be done by assigning a label to the adapter-name (e.g. adapter "ppp0" is to be assigned label L0) because the adapter names are created dynamically and the actual properties of the adapter may vary. By choosing a label appropriate to the adapter, it can be ensured that any security checks based on the label function properly. This is especially important with respect to Trusted Operating Systems (in particular, as defined with reference to the first and second aspects of the invention of our first co-pending International Application) which also apply labels to other system objects, such as processes, network connections, files, pipes, etc., in the sense that the label applied to the adapter has to be correct with respect to the other labels already present on the system.

The kernel/operating system typically has software-routines which are invoked when a new adapter is activated. In on exemplary embodiment of the invention of our second co-pending International Application, such routines are modified to also assign a label depending on the attributes of the newly-formed adapter, e.g. by consulting a ruleset or configuration table. Similarly, there are routines which are invoked when adapters are de-activated, which are modified to remove the label previously assigned.

Referring back to the first and second aspects of the invention of our first co-pending International Application, there is defined an operating system which augments each process and network interface with a tag indicating the compartment to which it belongs. In an exemplary embodiment, means provided in the kernel consult a rulebase whenever a process wishes to communicate with another process (in the Linux operating system, by using any of the standard UNIX inter-process communication mechanisms). The communication succeeds only if there is a matching rule in the rulebase. In the preferred embodiment, the rulebase resides in the kernel, but as explained above, to be more practical, it is preferably able to be initialized and dynamically maintained and queried by an administrative program, preferably in user-space.

Thus, the fifth aspect of the invention of our first co-pending International Application provides an operating system comprising a kernel including means for storing a rulebase consisting of one or more rules defining permitted communication paths between system objects, and user-operable means for adding, deleting and/or listing such rules.

Thus, in the operating system of the fifth aspect of the invention of our first co-pending International Application, it is possible to perform not just access control over TCP and UDP packets, but also other forms of inter-process communication that exist on the operating system (in a Linux system, these would include Raw IP packets, SysV messages, SysV shared memory and SysV semaphores).

In an exemplary embodiment of the fifth aspect of the invention of our first co-pending International Application, the user space program needs to be able to send and receive data from the kernel in order to change and list the entries in its rulebase. In a preferred

embodiment, this is implemented by the inclusion in the operating system of a kernel device driver which provides two entry points. The first entry point is for the 'ioctl' system call (ioctl is traditionally used to send small amounts of data or commands to a device. The first entry point is arranged to be used for three operations. Firstly, it can be used to specify a complete rule and add it to a rulebase. Secondly, the same data can be used to delete that rule. Thirdly, as an optimization, a rule can be deleted by its 'reference', which in one exemplary embodiment of the invention, is a 64-bit tag which is maintained by the kernel.

The second entry point is for a "/proc" entry. When the user space program opens this entry, it can read a list of rules generated by the kernel. The reason for this second entry point is that it is a more efficient mechanism by which to read the list of rules than via an ioctl command, and can be more easily read by other user processes which do not have to be specially written to recognize and handle the specific 'ioctl' commands for the kernel module.

Brief Description of the Drawings

FIGURE 1 is a schematic illustration of an exemplary architecture for multi-service hosting on an operating system with the containment property;

FIGURE 2 is a schematic illustration of an architecture of a trusted Linux host operating system according to an exemplary embodiment of the present invention;

FIGURE 3 illustrates an exemplary modified data type used in the operating system illustrated in Figure 2;

FIGURE 4 illustrates the major networking data types in Linux IP-networking;

FIGURE 5 illustrates the propagation of struct csecinfo data-members for IP-networking;

FIGURE 6 illustrates schematically three exemplary approaches to building containment into a Linux kernel;

FIGURE 7 illustrates schematically the effect of the rule;

HOST* -> COMPARTMENT x METHOD TCP PORT 80;

FIGURE 8 illustrates schematically the spectrum of options available for the construction of a hybrid containment prototype operating system;

FIGURE 9 illustrates schematically the desirability of updating replicated kernel

state in synchrony;

FIGURE 10 illustrates schematically an exemplary configuration of Apache and two Tomcat Java Vms;

FIGURE 11 illustrates schematically the layered chroot-ed environments in the Trusted Linux illustrated in Figure 2;

FIGURE 12 illustrates schematically the process of efficient lookup of kernel enforced compartment access control rules;

FIGURE 13 illustrates schematically an exemplary embodiment of a trusted gateway system according to an aspect of the present invention;

FIGURE 14 illustrates schematically the operation of an operating system according to an exemplary embodiment of an aspect of the present invention; and

FIGURE 15 illustrates schematically an exemplary embodiment of an operating system according to the prior art.

Detailed Description of the Invention

In summary, similar to the traditional trusted operating system approach, the property of containment is achieved in the operating system in an exemplary embodiment of the present invention by means of kernel level mandatory protection of processes, files and network resources. However, the mandatory controls used in the operating system of the present invention are somewhat different to those found on traditional trusted operating systems and, as such, they are intended to at least reduce some of the application integration and management problems associated with traditional trusted operating systems.

The key concept of a trusted operating system according to the invention is the 'compartment', and various services and applications on a system are run within separate compartments. Relatively simple mandatory access controls and process labeling are used to create the concept of a compartment. In the following exemplary embodiment of a trusted operating system according to the invention, each process within the system is allocated a label, and processes having the same label belong to the same compartment. Kernel level mandatory checks are enforced to ensure that processes from one compartment cannot interfere with processes from another compartment. The mandatory access controls are relatively simple

in the sense that labels either match or they do not. Further, there is no hierarchical ordering of labels within the system, as there is in some known trusted operating systems.

Unlike traditional trusted operating systems, in the present invention, labels are not used to directly control access to the main filesystem. Instead, filesystem protection is achieved by associating a different section of the main filesystem with each compartment. Each such section of the file system is a chroot of the main filesystem, and processes running within any compartment only have access to the section of filesystem which is associated with that compartment. Importantly, via kernel controls, the ability of a process to transition to root from within a compartment is removed so that the chroot cannot be escaped. An exemplary embodiment of the present invention also provides the ability to make at least selected files within a chroot immutable.

Flexible communication paths between compartments and network resources are provided via narrow, kernel level controlled interfaces to TCP/UDP plus most IPC mechanisms. Access to these communication interfaces is governed by rules specified by the security administrator on a 'per compartment' basis. Thus, unlike in traditional trusted operating systems, it is not necessary to override the mandatory access controls with privilege or resort to the use of user level trusted proxies to allow communication between compartments and network resources.

The present invention thus provides a trusted operating systems which offers containment, but also has enough flexibility to make application integration relatively straightforward, thereby reducing the management overhead and the inconvenience of deploying and running a trusted operating system.

The architecture and implementation of a specific exemplary embodiment of the present invention will now be described. In the following description, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art, that the invention may be practiced without limitation to these specific details. In other instances, well known methods and structures have not been described in detail so as to avoid unnecessarily obscuring the present invention.

In the following description, a trusted Linux operating system is described in detail, which system is realized by modification to the base Linux kernel to support containment of user-level services, such as HTTP-servers. However, it will be apparent to a person skilled in the art that the principles of the present invention could be applied to other types of operating system to achieve the same or similar effects.

The modifications made to a Linux operating system to realize a trusted operating system according to an exemplary embodiment of the invention, can be broadly categorized as follows:

1. Kernel modifications in the areas of:

   * TCP/IP networking
   * Routing-tables and routing-caches
   * System V IPC - Message queues, shared memory and semaphores
   * Processes and Threads
   * UID handling

2. Kernel configuration interfaces in the form of:

   * Dynamically loadable kernel modules
   * Command-line utilities to communicate with those modules

3. User-level scripts to administer/configure individual compartments:

   * Scripts to start/stop compartments

Referring to Figure 2 of the drawings, there is illustrated an architecture of a trusted Linux host operating system according to an exemplary embodiment of the invention, including the major areas of change to the base Linux kernel and the addition of a series of compartments in user-space implementing Web-servers capable of executing CGI-binaries in configurable chroot jails.

Thus, with reference to Figure 2, a base Linux kernel 100 generally comprises TCP/IP Networking means 102, UNIX domain sockets 104, Sys V IPC means 106 and other subsystems 108. The trusted Linux operating system additionally comprises kernel extensions 110 in the form of a security module 112, a device configuration module 114, a rule database 116 and kernel modules 118. As shown, at least some of the Linux kernel subsystems 102, 104, 106, 108 have been modified to make call outs to the kernel level security module 112. The security module 112 makes access control decisions and is responsible for enforcing the concept of a compartment, thereby providing containment.

The security module 112 additionally consults the rule database 116 when making a decision. The rule database 116 contains information about allowable communication paths between compartments, thereby providing narrow, well-controlled interfaces into and out of a compartment (see also Figure 12 of the drawings).

Figure 2 of the drawings also illustrates how the kernel extensions 110 are administered from user space 120 via a series of ioctl commands. Such ioctl commands take two forms: some to manipulate the rule table and others to run processes in particular compartments and configure network interfaces.

User space services, such as the web servers shown in Figure 2, are run unmodified on the platform, but have a compartment label associated with them via the command line interface to the security extensions. The security module 112 is then responsible for applying the mandatory access controls to the user space services based on their applied compartment label. It will be appreciated, therefore, that the user space services can thus be contained without having to modify those services.

The three major components of the system architecture described with reference to Figure 2 of the drawings are a) the command line utilities required to configure and administer the principal aspects of the security extensions, such as the communication rules and process compartment labels; b) the loadable modules that implement this functionality within the kernel; and c) the kernel modifications made to take advantage of this functionality. These three major components will now be described in more detail, as follows.

a) Command-line Utilities

'CACC' is a command line utility to add, delete and list rules via /dev/cacc and /proc/cacc interfaces provided by a cac kernel-loadable module (not shown). Rules can either be entered on the command line, or can be read from a text-file.

In this exemplary embodiment of the invention, rules take the following format:

<rule>::=<source>[<port>]-><destination>[<port>]<method list><netdev>

where:

<identifier> == (<compartment> | <host> | <net>) [<port>]
<compartment> == 'COMPARTMENT' <comp_name>
<host> == 'HOST' <host_name>
<net> == 'NET' <ip_addr> <netmask>
<net> == 'NET' <ip_addr> '/' <bits>

<comp_name> == A valid name of a compartment
<host_name> == A known hostname or IP address
<ip_addr> == An IP address in the form a.b.c.d
<netmask> == A valid netmask, in the form a.b.c.d
<bits> == The number of leftmost bits in the netmask.... 0 thru 31
<method_list> == A list of comma-separated methods (In this exemplary embodiment, methods supported are: TCP (Transmission Control Protocol), UDP

(User Datagram Protocol), and ALL.

To add a rule, the user can enter 'cacc -a <filename>'(to read a rule from a text file, where <filename> is a file containing rules in the format described above), or 'cacc -a rule' (to enter a rule on the command line).

To delete a rule, the user can enter 'cacc - d <filename>', or cacc -d rule, or cacc -d ref (in this form, a rule can be deleted solely by its reference number which is output by listing the rules using the command cacc -l, which outputs or lists the rules in a standard format with the rule reference being output as a comment at the end of each rule.

By default, 'cacc' expects to find the compartment mapping file 'cmap.txt' and the method mapping file 'mmap.txt' in the current working directory. This can be overridden, however, by setting the UNIX environment variables CACC_CMAP and CACC_MMAP to where the files actually reside, in this exemplary embodiment of the invention.

Any syntax or semantic errors detected by cacc will cause an error report and the command will immediately finish, and no rules will be added or deleted. If a text file is being used to enter the rules, the line number of the line in error will be found in the error message.

Another command-line utility provided by this exemplary embodiment of the present invention is known as 'lcu', which provides an interface to an LNS kernel-module (not shown). Its most important function is to provide various administration-scripts with the ability to spawn processes in a given compartment and to set the compartment number of interfaces. Examples of its usage are:

1. 'lcu setdev eth0 0xFFFF0000'
   Sets the compartment number of the eth0 network interface to 0xFFFF0000

2. 'lcu setprc 0x2 -cap_mknod bash'
   Switches to compartment 0x2, removes the cap_mknod capability and invokes bash

b) Kernel Modules

This exemplary embodiment of the present invention employs two kernel modules to implement custom ioctl()s that enable the insertion/deletion of rules and other functions such as labeling of network interfaces. However, it is envisaged that the two modules could be merged and/or replaced with custom system-calls. In this embodiment of the present invention, the two kernel modules are named *lns* and *cac*.

The lns module implements various interfaces via custom ioctl()s to enable:

1. A calling process to switch compartments.
2. Individual network interfaces to be assigned a compartment number.
3. Utility functions, such as process listing with compartment numbers and the logging of activity to kernel-level security checks.

The main client of this module is the *lcn* command-line utility described above.

The cac module implements an interface to add/delete rules in the kernel via a custom ioctl(). It performs the translation between higher-level simplified rules into primitive forms more readily understood by kernel lookup routines. This module is called by the *cacc* and *cgicacc* user-level utilities to manipulate rules within the kernel.

c) Kernel Modifications

In this exemplary embodiment of the present invention, modifications have been made to the standard Linux kernel sources so as to introduce a tag on various data types and for the addition of access-control checks made around such tagged data types. Each tagged data type contains an additional struct csecinfo data-member which is used to hold a compartment number (as shown in Figure 3 of the drawings). It is envisaged that the tagged data types could be extended to hold other security attributes. In general, the addition of this data-member is usually performed at the very end of a data-structure to avoid issues arising relating to the common practice casting pointers between two or more differently named structures

which begin with common entries.

The net effect of tagging individual kernel resources is to very simply implement a compartmented system where processes and the data they generate/consume are isolated from one another. Such isolation is not intended to be strict in the sense that many covert channels exist (see discussion about processes below). The isolation is simply intended to protect obvious forms of conflict and/or interaction between logically different groups of processes.

In this exemplary embodiment of the present invention, there exists a single function cnet_chk_attr() that implements a yes/no security check for the subsystems which are protected in the kernel. Calls to this function are made at the appropriate points in the kernel sources to implement the compartmented behavior required. This function is predicated on the subsystem concerned and may implement slightly different defaults or rule-conventions depending on the subsystem of the operation being queried at that time. For example, most subsystems implement a simple partitioning where only objects/resources having exactly the same compartment number result in a positive return value. However, in certain cases, the use of a no-privilege compartment 0 and/or a wildcard compartment -1L can be used, e.g. compartment 0 as a default 'sandbox' for unclassified resources/services; a wildcard compartment for supervisory purposes, like listing all processes on the subsystem prior to shutting down.

Referring to Figure 4 of the drawings, standard Linux IP networking will first be explained. Each process or thread is represented by a task_struct variable in the kernel. A process may create sockets in the AF_INET domain for network communication over TCP/UDP. These are represented by a pair of struct socket and struct sock variables, also in the kernel.

The struct sock data type contains, among other things, queues for incoming packets represented by struct sk_buffs. It may also hold queues for pre-allocated sk_buffs for packet transmission. Each sk_buff represents an IP packet and/or fragment traveling up/down the IP stack. They either originate at a struct sock (or, more specifically, from its internally pre-allocated send-queue) and travel downwards for transmission, or they originate from a network driver and travel upwards from the bottom of the stack starting from a struct

net_device which represents a network interface. When traveling downwards, they effectively terminate at a struct net_device. When traveling upwards, they are usually delivered to a waiting struct sock (actually, its pending queue).

Struct sock variables are created essentially indirectly by the socket()-call (in fact, there are private per-protocol sockets owned by various parts of the stack within the kernel itself that cannot be traced to a running process), and can usually be traced to an owning user-process, i.e. a task_struct. There exists a struct net_device variable for each configured interface on the system, including the loopback interface. Localhost and loopback communications appear not to travel via a fastpath across the stack for speed, instead they travel up and down the stack as would be expected for remote host communications. At various points in the stack, calls are made to registered netfilter-modules for the purposes of packet interception.

By adding an additional csecinfo data-member to the most commonly used data types in Linux IP networking, it becomes possible to trace ownership and hence read/write dataflows of individual IP packets for all running processes on the system, including kernel-generated responses.

Thus, in order to facilitate this exemplary embodiment of the present invention, at least the major networking data types used in standard Linux IP networking have been modified. In fact, most of the data-structures modified to realize this embodiment of the invention are related to networking and occur in the networking stack and socket-support routines. The tagged network data structures serve to implement a partitioned IP stack. In this exemplary embodiment of the invention, the following data structures have been modified to include a struct csecinfo:

1. struct task_struct - processes (and threads)
2. struct socket - abstract socket representation
3. struct sock - domain-specific socket
4. struct sk_buff - IP packets or messages between sockets
5. struct net_device - network interfaces, e.g. eth0, lo, etc.

During set-up, once the major data types were tagged, the entire IP-stack was checked for points at which these data types were used to introduce newly initialized variables into the kernel. Once such points had been identified, code was inserted to ensure that the inheritance of the csecinfo structure was carried out. The manner in which the csecinfo structure is propagated throughout the IP networking stack will now be described in more detail.

There are two named sources of struct csecinfo data members, namely per-process task_structs and per-interface net_devices. Each process inherits its csecinfo from its parent, unless explicitly modified by a privileged ioctl(). In this exemplary embodiment of the present invention, the init-process is assigned a compartment number of 0. Thus, every process spawned by init during system startup will inherit this compartment number, unless explicitly set otherwise. During system startup, init-scripts are typically called to explicitly set the compartment numbers for each defined network interface. Figure 5 of the drawings illustrates how csecinfo data-members are propagated for the most common cases.

All other data structures inherit their csecinfo structures from either a task_struct or a net_device. For example, if a process creates a socket, a struct socket and/or struct sock may be created which inherit the current csecinfo from the calling process. Subsequent packets generated by calling write() on a socket generate sk_buffs which inherit their csecinfo from the originating socket.

Incoming IP packets are stamped with the compartment number of the network interface on which it arrived, so sk_buffs traveling up the stack inherit their csecinfo structure from the originating net_device. Prior to being delivered to a socket, each sk_buff's csecinfo structure is checked against that of the prospective socket.

It will be appreciated that special care must be taken in the case of non-remote networking, i.e. in the case where a connection is made between compartments X and Y through any one of the number of network interfaces which is allowed by a rule of the form:

COMPARTMENT X -> COMPARTMENT Y METHOD tcp

Because the security checks occur twice for IP networking, i.e. once on output and once on input, it is necessary to provide means for preventing the system from looking for the existence of these rules instead:

COMPARTMENT X -> HOST a.b.c.d METHOD tcp (for output)
HOST a.b.c.d -> COMPARTMENT Y METHOD tcp (for input)

which, although valid, may not be used in preference to the rule specifying source and destination compartments directly. To cater for this, in this exemplary embodiment of the invention, packets sent to the loopback device retain their original compartment numbers and are simply 'reflected' off it for eventual delivery. Note that, in this case, the security check occurs on delivery and not transmission. Upon receipt of an incoming local packet on the loopback interface, the system is set up to avoid overwriting the compartment number of the packet with that of the network interface and allow it to travel up the stack for the eventual check on delivery. Once there, the system performs a check for a rule of the form:

COMPARTMENT X -> COMPARTMENT Y tcp

instead of

HOST a.b.c.d -> COMPARTMENT Y METHOD tcp

because of the presence on the sk_buff of a compartment number that is not of a form normally allocated to network interfaces (network interfaces in this exemplary embodiment of the present invention, as a general rule, are allocated compartment numbers in the range 0xFFFF0000 and upwards and can therefore be distinguished from those allocated for running services).

Because the rules are unidirectional, the TCP layer has to dynamically insert a rule to handle the reverse data flow once a TCP connection has been set up, either as a result of a connect() or accept(). This happens automatically in this exemplary embodiment of the invention and the rules are then deleted once the TCP connection is closed. Special handling occurs when

a struct tcp_openreq is created to represent the state of a pending connection request, as opposed to one that has been fully set up in the form of a struct sock. A reference to the reverse-rule created is stored with the pending request and is also deleted if the connection request times out or fails for some other reason.

An example of this would be when a connection is made from compartment 2 to a remote host 10.1.1.1. The original rule allowing such an operation might have looked like this:

COMPARTMENT 2 -> NET 10.1.1.0/255.255.255.0 METHOD tcp

As a result, the reverse rule would be something like this (abc/xyz being the specific port-numbers used):

HOST 10.1.1.1 PORT abc -> COMPARTMENT 2 PORT xyz METHOD tcp

In order to support per-compartment routing-tables, each routing table entry is tagged with a csecinfo structure. The various modified data structures in this exemplary embodiment of the invention are:

1. struct rt_key
2. struct rtable
3. struct fib_rule
4. struct fib_node

Inserting a route using the route-command causes a routing-table entry to be inserted with the csecinfo structure inherited from the calling context of the user-process, i.e. if a user invokes the route-command from a shell in compartment N, the route added is tagged with N as the compartment number. Attempts to view routing-table information (usually by inspecting /proc/net/route and /proc/net/rt_cache) are predicated on the value of the csecinfo structure of the calling user-process.

The major routines used to determine input and output routes which a sk_buff should take are

ip_route_output() and ip_route_input(). In this exemplary embodiment of the invention, these have been expanded to include an extra argument consisting of a pointer to the csecinfo structure on which to base any routing-table lookup. This extra argument is supplied from either the sk_buff of the packet being routed for input or output.

Kernel-inserted routing-entries have a special status and are inserted with a wildcard compartment number (-1L). In the context of per-compartment routing, they allow these entries to be shared across all compartments. The main purpose of such a feature is to allow incoming packets to be routed properly up the stack. Any security-checks occur at a higher level just prior to the sk_buff being delivered on a socket (or its sk_buff queue).

The net effect is that each compartment appears to have their individual routing tables which are empty by default. Every compartment shares the use of system-wide network-interfaces. In this exemplary embodiment of the invention, it is possible to restrict individual compartments to a strict subset of the available network-interfaces. This is because each network-interface is notionally in a compartment of its own (with its own routing table). In fact, to respond to an ICMP-echo request, each individual interface can optionally be configured with tagged routing-table entries to allow the per-protocol ICMP-socket to route its output packet.

Other Subsystems

* UNIX Domain Sockets - Each UNIX domain socket is also tagged with the csecinfo structure. As they also use sk_buffs to represent messages/data traveling between connected sockets, many of the mechanisms used by the AF_INET domain described above apply similarly. In addition, security-checks are also performed at every attempt to connect to a peer.

* System V IPC - Each IPC-mechanism listed above is implemented using a dedicated kernel structure that is similarly tagged with a csecinfo structure. Attempts to list, add or remove messages to these constructs are subject to the same security checks as individual sk_buffs. The security checks are dependent on the exact type of mechanism used.

- Processes/Threads - Since individual processes, i.e. task_structs are tagged with the csecinfo structure, most process-related operations will be predicated on the value of the process's compartment number. In particular, process listing (via the /proc interface) is controlled as such to achieve the effect of a per-compartment process-listing. Signal-delivery is somewhat more complicated as there are issues to be considered in connection with delivery of signals to parent processes which may have switched compartments - thus constituting a 1-bit covert channel.

System Defaults

Per-protocol Sockets - The Linux IP stack uses special, private per-protocol sockets to implement various default networking behaviors such as ICMP-replies. These per-protocol sockets are not bound to any user-level socket and are typically initialized with a wildcard compartment number to enable the networking functions to behave normally.

Use of Compartment 0 as Unprivileged Default - The convention is to never insert any rules which allow Compartment 0 any access to other compartments and network-resources. In this way, the default behavior of initialized objects, or objects which have not been properly accounted for, will fall under a sensible and restricted default.

Default Kernel Threads - Various kernel threads may appear by default, e.g. kswapd, kflushd, and kupdate to name but a few. These threads are also assigned a csecinfo structure per-task_struct and their compartment numbers default to 0 to reflect their relatively unprivileged status.

Sealing Compartments against Assumption of Root-identity - Individual compartments may optionally be registered as 'sealed' to protect against processes in that compartment from successfully calling setuid(0) and friends, and also from executing any SUID-root binaries. This is typically used for externally-accessible services which may in general be vulnerable to buffer-overflow attacks leading to the execution of malicious code. If such services are constrained to being initially run as a pseudo-user (non-root) and if the compartment it executes in is sealed, then any attempt to assume the root-identity either by buffer-overflow

attacks and/or execution of foreign instructions will fail. Note that any existing processes running as root will continue to do so.

The kernel modifications described previously serve to support the hosting of individual user-level services in a protected compartment. In addition to this, the layout, location and conventions used in adding or removing services in this exemplary embodiment of the invention will now be described.

Individual services are generally allocated a compartment each. However, what an end-user perceives as a service may actually end up using several compartments. An example would be the use of a compartment to host an externally-accessible Web-server with a narrow interface to another compartment hosting a trusted gateway agent for the execution of CGI-binaries in their own individual compartments. In this case, at least three compartments would be needed:

* one for the web-server processes;
* one for the trusted gateway agent which executes CGI-binaries; and
* as many compartments as are needed to properly categorize each CGI binary, as the trusted gateway will fork/exec CGI-binaries in their configured compartments.

Every compartment has a name and resides as a chroot-able environment under /compt. Examples used in an exemplary embodiment of the present invention include:

Location	Description
/compt/admin	Admin HTTP-server
/compt/omailout	Externally visible HTTP-server hosting OpenMail server processes
/compt/omailin	Internal compartment hosting OpenMail server processes
/compt/web1	Externally visible HTTP-server

/compt/web1mega	Internal Trusted gateway agent for Web1's CGI-binaries

In addition, the following subdirectories also exist:

1. /compt/etc/cac/bin - various scripts and command-line utilities for managing compartments
2. /compt/etc/cac/rules - files containing rules for every registered compartment on the system
3. /compt/etc/cac/encoding - configuration file for the cacc-utility, e.g. compartment-name mappings

To support the generic starting/stopping of a compartment, each compartment has to conform to a few basic requirements:

1. be chroot-able under its compartment location /compt/<name>
2. provide /compt/<name>/startup and /compt/<name>/shutdown to start/stop the compartment
3. startup and shutdown scripts are responsible for inserting rules, creating routing-tables, mounting filesystems (e.g. /proc)and other per-service initialization steps

In general, if the compartment is to be externally visible, the processes in that compartment should not run as root by default and the compartment should be sealed after initialization. Sometimes this is not possible due to the nature of a legacy application being integrated/ported, in which case it is desirable to remove as many capabilities as possible in order to prevent the processes from escaping the chroot-jail, e.g. cap_mknod.

Due to the fact that the various administration scripts require access to each configured compartment's filesystem, and that these administration-scripts are called via the CGI-

interface of the administration Web-server, it is the case that these scripts cannot reside as a normal compartment, i.e. under /compt/<name>.

In this exemplary embodiment of the invention, the approach taken is to enclose the chroot-able environment of the administration scripts around every configured compartment, but to ensure that the environment is a strict subset of the host's filesystem. The natural choice is to make the chroot-jail for the administration scripts to have its root at/compt. The resulting structure is illustrated schematically in Figure 11 of the drawings.

Since compartments exist as chroot-ed environments under the /comp directory, application-integration requires the usual techniques used for ensuring that they work in a chroot-ed environment. A common technique is to prepare a cpio-archive of a minimally running compartment, containing a minimal RPM-database of installed software. It is usual to install the desired application on top of this and, in the case of applications in the form of RPM's, the following steps could be performed:

root@tlinux#	chroot /compt/app1
root@tlinux#	rpm -install <RPM-package-filename>
root@tlinux#	[Change configuration files as required, e.g. httpd.conf]
root@tlinux#	[Create startup/shutdown scripts in /compt/app1]

The latter few steps may be integrated into the RPM-install phase. Reductions in disk-space can be achieved by inspection: selectively uninstalling unused packages via the rpm-command. Additional entries in the compartment's /dev-directory may be created if required, but /dev is normally left substantially bare in most cases. Further automation may be achieved by providing a Web-based interface to the above-described process to supply all of the necessary parameters for each type of application to be installed. No changes to the compiled binaries are needed in general, unless it is required to install compartment-aware variants of such applications.

A specific embodiment of one aspect of the present invention has been described in detail above. However, a variety of different techniques may be used in the implementation of the

general concept of containment provided by the present invention. It is obviously undesirable to rewrite the operating system because it is necessary to be able to reuse as many user-level applications as possible. This leaves various interposition techniques, some of which are listed below, and can be categorized as either primarily operating at the user-level or kernel-based.

User-level techniques

The following outlines three common user-level techniques or mechanisms.

1. The strace() mechanism

This mechanism uses the functionality built into the system kernel to trace each system-call of a chosen process. Using this mechanism, each system-call and its arguments can be identified and the system-call is usually either allowed to proceed (sometimes with modified arguments) or to fail according to a defined security policy.

This mechanism, while suitable for many applications, has a number of drawbacks. One of these drawbacks becomes apparent in the case of the 'runaway child' problem, in which a process P which is being traced may fork a child Q which is scheduled to run before P returns from the fork() system-call. Since strace() works by attaching to processes using process ID's (PID's), and the PID of Q is not necessarily returned to P (and hence the tracer) before Q is actually scheduled to run, there is a risk that Q would be allowed to execute some arbitrary length of code before the tracer can be attached to it.

One solution to this problem is to check every system-call in the kernel for as-yet untraced processes and to trap them there, for example, by forcefully 'putting them to sleep' so that the tracer can eventually catch up with them. This solution would, however, require an additional kernel component.

2. System-call wrapping

Another drawback of this mechanism occurs in the case that there exists a race-condition where arguments to a traced system-call can be modified. The window where this occurs happens between the tracer inspecting the set of arguments and actually allowing the system call to proceed. A thread sharing the same address-space as the traced process can modify the arguments in-memory during this interval.

Using this mechanism, system-calls can be wrapped using a dynamically linked shared library that contains wrappers to system-calls that are linked against a process which is required to be trace. These wrappers could contain call-outs to a module that makes a decision according to a predefined security policy.

One drawback associated with this mechanism is that it may be easily subverted if the system-calls that a process presumes to use are not unresolved external references and cannot be linked by the dynamic loader. It is also possible to make a system-call that by-passes the wrapper if the process performs the soft-interrupt itself with the correct registers set up like a normal system-call. In this case, the kernel handles the call without passing through a wrapper. In addition, in some cases, the dependence on the LD_PRELOAD environment variable might also be an unacceptable weak link.

3. User-level authorization servers

This category includes authorization servers in user-space acting on data supplied via a private channel to the kernel. Although very effective in many cases, this approach does have a number of disadvantages, namely I) each system-call being checked incurs at least two context-switches, making this solution relatively slow; ii) interrupt routines are more difficult to bridge into user-space kernels due to the requirement that they do not sleep; and iii) a kernel-level component is usually required to enforce mandatory tracing.

Despite the disadvantages of the user-level approaches outlined above, user-level techniques to implement a trusted operating system in accordance with one aspect of the present invention have the advantage of being relatively easy to develop and maintain, although in some circumstances they may be insufficient in the implementation of system-wide mandatory

controls.

Ultimately, the aim of the present invention is to contain running applications, preferably implemented by a series of mandatory access controls which cannot be overridden on a discretionary basis by an agent that has not been authorized directly by the security administrator. Implementing containment in a fashion that is transparent to running third-party applications can be achieved by kernel-level access controls. By examining the possible entry points and separating out the interactions of the kernel subsystems within and against each other, it becomes possible to segment the view of the kernel and its resources with respect to the running applications.

Such a scheme of segmentation is mandatory in nature due to its implementation within the kernel itself - there is no discretionary aspect that can be overridden by a running application unless it is made explicitly aware of the containment scheme and has been re-written to take advantage of it.

Three examples of kernel-level approaches to implementing the present invention are outlined below and illustrated in Figure 6 of the drawings. The first approach is based primarily on patches to the kernel and its internal data structures. The second approach is entirely different in that it does not require any kernel patches at all, instead being a dynamically loadable kernel module that operates by replacing selected system calls and possibly modifying the run-time kernel image. Both of these approaches require user-level configuration utilities typically operating via a private channel into the kernel. The third approach represents a compromise between the absolute controls offered by the first approach versus the independence from kernel-source modifications offered by the second.

1. Source-level Kernel Modifications to Support Containment (V1)

This approach is implemented as a series of patches to standard operating system (in this case, Linux) kernel sources. There is also a dynamically loadable kernel module that hosts the logic required to maintain tables of rules an also acts as an interface between the kernel and user-space configuration utilities. The kernel module is inserted early in the boot-sequence and

immediately enforces a restrictive security model in the absence of any defined rules. Prior to this, the kernel enforces a limited security model designed to allow proper booting with all processes being spawned in the default compartment 0 that is functional but essentially useless for most purposes. Once the kernel module is loaded, the kernel switches from its built-in model to the one in the module. Containment is achieved by tagging kernel resources and partitioning access to these depending on the value of the tags and any rules which may have been defined.

Thus, each kernel resource required to be protected is extended with a tag indicating the compartment that the resource belongs to (as described above). A compartment is represented by a single word-sized value within the kernel, although more descriptive string names are used by user-level configuration utilities. Examples of such resources include data-structures describing:

* individual processes
* shared-memory segments
* semaphores, message queues
* sockets, network packets, network-interfaces and routing-table enquiries

A complete list of modified data structures to support this approach to containment according to an exemplary embodiment of the invention is given in Appendix 7.1 attached hereto. As explained above, the assignment of the tag occurs largely through inheritance, with the *init*-process initially being assigned to compartment 0. Any kernel objects created by a process inherit the current label of the running process. At appropriate points in the kernel, access-control checks are performed through the use of hooks to a dynamically loadable security-module that consults a table of rules indicating which compartments are allowed to access the resources of another compartment. This occurs transparently to the running applications.

Each security check consults a table of rules. As described above, each rule has the form:

source -> destination method m [attr]
[netdev n]

where:

source/destination is one of:

COMPARTMENT (a named compartment)

HOST (a fixed IPv4 address)

NETWORK (an IPv4 subnet)

m: supported kernel mechanism, e.g. tcp, udp, msg (message queues), shm (shared-memory), etc.

attr: attributes further qualifying the method m

n: a named network-interface if applicable, e.g. eth0

An example of such a rule which allows processes in the compartment named "WEB" to access shared-memory segments, for example using *shmat/shmdt()*, from the compartment named "CGI" would look like:

COMPARTMENT:WEB -> COMPARTMENT:CGI METHOD shm

Present also are certain implicit rules, which allow some communications to take place within a compartment, for example, a process might be allowed to see the process identifiers of processes residing in the same compartment. This allows a bare-minimum of functionality within an otherwise unconfigured compartment. An exception is compartment 0, which is relatively unprivileged and where there are more restrictions applied. Compartment 0 is typically used to host kernel-level threads (such as the swapper).

In the absence of a rule explicitly allowing a cross-compartment access to take place, all such attempts fail. The net effect of the rules is to enforce mandatory segmentation across individual compartments, except for those which have been explicitly allowed to access another compartment's resources.

The rules are directional in nature, with the effect that they match the connect/accept behavior of TCP socket connections. Consider a rule used to specify allowable incoming HTTP connections of the form:

HOST* -> COMPARTMENT X METHOD TCP PORT 80

This rule specifies that only incoming TCP connections on port 80 are to be allowed, but not outgoing connections (see Figure 7). The directionality of the rules permits the reverse flow of packets to occur in order to correctly establish the incoming connection without allowing outgoing connections to take place.

The approach described above has a number of advantages. For example, it provides complete control over each supported subsystem and the ability to compile out unsupported ones, for example, hardware-driven card-to-card transfers. Further, this approach provides relatively comprehensive namespace partitioning, without the need to change user-space commands such as *ps, netstat, route, ipcs* etc. Depending on the compartment that a process is currently in, the list of visible identifiers changes according to what the rules specify. Examples of namespaces include Process-table via/proc, SysV IPC resource-identifiers, Active, closed and listening sockets (all domains), and Routing table entries.

Another advantage of this approach is the synchronous state with respect to the kernel and its running processes. In view of the fact that the scalar tag is attached to the various kernel-resources, no complete lifetime tracking needs to be done which is a big advantage when considering the issue of keeping the patches up to date as it requires a less in-depth understanding of where kernel variables are created/consumed. Further, fewer source changes need to be made as the inheritance of security tags happens automatically through the usual C assignment-operator (=) or through *memcpy()*, instead of having to be explicitly specified through the use of #ifdefs and clone-routines.

In addition, there is no need to recursively enumerate kernel resources at the point of activation as such accounting is performed the moment the kernel starts. Further, this approach provides a relatively speedy performance (about 1 - 2 % of optimal) due to the relatively small number of source changes to be made. Depending on the intended use of the system, the internal hash-tables can be configured in such a way that the inserted rules are on average 1-level deep within each hash-bucket - this makes the rule-lookup routines behave in the order of O(1).

However, despite the numerous advantages, this approach does require source modifications to the kernel, and the patches need to be updated as new kernel revisions become available. Further, proprietary device-drivers distributed as modules cannot be used due to possible structure-size differences.

2. System-call Replacement via Dynamically Loadable Kernel Modules (V2)

This approach involves implementing containment in the form of a dynamically loadable kernel module and represents an approach intended to recreate the functionality of the Source-level Kernel Modification approach outlined above, without needing to modify kernel sources.

In this approach, the module replaces selected system-calls by overwriting the *sys_call_table[]* array and also registers itself as a *netfilter* module in order to intercept incoming/outgoing network packets. The module maintains process ID (PID) driven internal state-tables which reflect the resources claimed by each running process on the system, and which are updated at appropriate points in each intercepted system call. These tables may also contain security attributes on either a per-process or per-resource basis depending on the desired implementation.

The rule format and syntax for this approach is substantially as described with regard to the Source-level Kernel Modification approach outlined above, and behaves in a similar manner. Segmentation occurs through the partitioning of the namespaces at the system-call layer. Access to kernel resources via the original system-calls becomes conditional upon security checks performed prior to making the actual system call.

All system-call replacements have a characteristic pre/actual/post form to reflect the conditional nature of how system-calls are handled in this approach.

Thus, this approach has the advantage that no kernel modifications are required, although knowledge of the kernel internals is needed. Further, the categorization of bugs becomes easier with the ability to run the system while the security module is temporarily disabled.

There are also a number of disadvantages and/or issues to be considered in connection with this approach. Firstly, maintaining true synchronous state with respect to the running processes is difficult for various reasons that are mostly due to the lack of a comprehensive kernel event notification mechanism. For example, there is no formal mechanism for catching the situation where processes exit abnormally, e.g. due to SIGSEGV, SIGBUS, etc. One proposed solution to this problem involves a small source code modification to *do_exit()* to provide a callback to catch such cases. In one exemplary embodiment, a kernel-level reaper thread may be used to monitor the global tasklist and perform garbage collecting on dead PID's. This introduces a small window of insecurity which is somewhat offset by the fact that PID's cycle upwards and the possibility of being reassigned a previously used PID within a single cycle of the reaper thread is relatively small.

With regard to the runaway-child problem described above, fork/vfork/clone does not return with the child's PID until possibly after the child is scheduled to run. If the module implementation creates PID-driven state-tables, this means that the child may invoke system-calls prior to a state-entry being created for it. The same problem exists in the *strace* command (as described above) which cannot properly follow forked children due to the need to attach to child processes. One possible solution to this problem is to intercept all system-calls with pre-conditional checks, but this solution is relatively slow and ineffective in some circumstances.

Another possible solution is relatively complex, and illustrated in Appendix 7.2 attached hereto.

1. *fork()* - the return address on the stack of the parent is modified prior to calling the real fork()-system call by poking the stack in the user-space. This translates to the child inheriting the modified return address. The modified return address is set to point to 5 bytes prior to its original value which causes the *fork()* system call to be called again by the child as its first action. The system then intercepts this and creates the necessary state entries. The parent has the saved return-address restored just prior to returning from *fork()* and so proceeds as normal. (Note that 5 bytes is exactly the length of the instruction for a form of the IA-32 far call. Other variants may be wrapped using LD_PRELOAD and a syscall wrapper that has

the desired 5-byte form).

2. *clone()* - the method used for a forked child (as described above) is not suitable for handling a cloned child due to the different way the stack is set up. The proposed solution instead is to:

a. Call *brk()* on behalf of the user-process to allocate a small 256-byte chunk of memory;

b. Copy a prepared chunk of executable code into this newly-allocated memory. This code will call a designated system-call before proceeding as normal for a cloned child;

c. Modify the stack of the user-process so that it executes this newly-prepared chunk of code instead of the original routine supplied in the call to *clone()*;

d. Save the original pointer to the routine supplied by the user-process to clone.

When the cloned child first executes, it will run the prepared chunk of code that makes a system-call which returns the pointer to the original routine that it was supposed to have executed. The child is trapped at this point and state-entries are created for it. The cloned child then executes the original routine as normal. (See Appendix 7.4 attached hereto).

In both cases, the child is forcibly made to call down to the kernel-module where it can be trapped.

Another possible solution is to change the *ret_from_fork()* routine in the kernel to provide a callback each time a child is created. Alternatively, the *do_fork()* kernel function which implements fork/vfork/clone could be modified.

Tracking close-on-exec behavior is also difficult in this implementation without intimate knowledge of the filesystem-related structures within each process structure.

Another issue to be considered in connection with this approach is that the module should

typically be loaded very early in the boot sequence to start monitoring kernel resources as soon as possible because post-enumerating such resources becomes progressively more difficult as the boot sequence advances. It should also be noted that the process of checking for the validity of system-call arguments in this approach is shifted to the kernel module instead of the original system-calls. As such, because the original kernel is not modified, additional overhead is introduced with this approach. Similarly, maintaining what is essentially replicated state information apart from the kernel adds overhead in terms of memory usage and processor cycles.

Yet another disadvantage is the loss of per-compartment routing and the features that depend on it, namely virtualized ARP caches and the ability to segment back-end network access using routes. This is because the routing code is run unmodified without tagged data structures. Finally, it is considered very difficult, if not impossible, to provide a single binary module that caters to all configurations. The size and layout of data-members within a structure depend on the config-options in that particular kernel-build. For example, specifying that *netfilter* be compiled causes some networking-related data structures to change in size and layout.

There are a number of issues to be considered in connection with the deployment of the dynamically loadable kernel module. Because the size of certain kernel data structures depends on the actual configuration options determined at build-time, i.e. the number of data members can vary depending on what functionality has been selected to be compiled in the kernel, the need to match the module to the kernel is essential. Thus, modules can either be built against known kernels, in which case, the sources and the configuration options (represented by a config-file) is readily available, or modules can be built at the point of installation, in which case the sources to the module would have to be shipped to the point of installation.

3. Hybrid System-call Replacement with Support from Kernel-based Changes

Referring to Figure 8 of the drawings, there is illustrated schematically some of the options available for the construction of a hybrid containment operating system which combines some

of the features of the modified kernel-based approach (V1) and the system-call replacement approach (V2) as described above.

In terms of maintaining state relative to the running kernel, the V1 approach is much more closely in step with the actual operation of the kernel compared to V2, which remains slightly out of step due to the lack of proper notification mechanisms and the need for garbage collecting. The state information in V1 is *synchronous* with respect to the kernel proper, and V2 is *asynchronous*. Synchrony is determined by whether or not the internal state-tables are updated in lock-step fashion with changes in the actual kernel state, typically within the same section of code bounded by the acquisition of synchronization primitives. The need for synchrony is illustrated in Figure 9 of the drawings, where changes to kernel state arising from an embedded source need to be reflected in the replicated state at the interposition layer.

Referring back to Figure 8 of the drawings, the determination of relative advantages in connection with the V1 and V2 approaches works on a sliding scale between the position of synchronous state typified by the V1 approach and the asynchronous one offered by the V2 approach, depending on how aggressively a developer wishes to modify kernel sources in order to achieve a near-synchronous state. Figure 8 illustrates three points at which changes to the V2 approach might provide significant advantages at the relatively slight expense of kernel source code changes.

1. *do_exit()* - a 5-line change in the *do_exit()* kernel function would enable a callback to be provided to catch changes to the global tasklist as a result of processes terminating abnormally. Such a change does not require knowledge of how the process termination is handled, but an understanding of where the control paths lie.

2. Fork/vfork/clone - another 5-line change in the do_fork kernel function would allow the proper notification of child PID's before they can be scheduled to run. An alternative is to modify *ret_from_fork()* but this is architecture-dependent. Neither of these options requires knowledge of process setup, just an awareness of the nature of PID creation and the locks surrounding the PID-related structures.

3. Interrupts, TCP timers, etc. - this category covers all operations carried out asynchronously in the kernel as a result of either a hard/soft IRQ, tasklets, internal timers or any execution context not traceable to a user-process. An example is the TCP timewait hash buckets used to maintain sockets that have been closed, but are yet to disappear completely. The hashtables are not publicly exported and changes to them cannot be tracked, as there are no formal API's for callbacks. If it is required to perform accounting on a per-packet basis (which is a major advantage in the V1 approach and from which several features are derived), then this category of changes to the kernel sources is required. However, in order to carry out those (relatively extensive) changes, an in-depth knowledge of the inner workings of the subsystems involved.

One of the most important applications of the present invention is the provision of a secure web server platform with support for the contained execution of arbitrary CGI-binaries and with any non-HTTP related processing (e.g. Java servlets) being partitioned into separate compartments, each with the bare minimum of rules required for their operation. This is a more specific configuration than the general scenario of:

1. Secure gateway systems which host a variety of services, such as DNS, Sendmail, etc. Containment or compartmentalization in such systems could be used to reduce the potential for conflict between services and to control the visibility of back-end hosts on a per-service basis.

2. Clustered front-ends (typically HTTP) to multi-tiered back-ends, including intermediate application servers. Compartmentalization in such systems has the desired effect of factoring out as much code as possible that is directly accessible by external clients.

In summary, the basic principle behind the present invention is to reduce the size and complexity of any externally accessible code to a minimum, which restricts the scope by which an actual security breach may occur. The narrowest of interfaces possible are specified between the various functional components which are grouped into individual compartments by using the most specific rule possible and/or by taking advantage of the directionality of the rules.

Returning now to Figure 2 of the drawings, there is illustrated a web-server platform which is configured based on V1 as the chosen approach. As described above, each web-server is placed in its own compartment. The MCGA daemon handles CGI execution requests and is placed in its own compartment. There are additional compartments for administration purposes as well. Also shown is the administration CGI utilities making use of user-level command line utilities to configure the kernel by the addition/deletion of rules and the setting of process labels. These utilities operate via a privileged device-driver interface. In the kernel, each subsystem contains call-outs to a custom security module that operates on rules and configuration information set earlier. User-processes that make system calls will ultimately go through the security checks present in each subsystem and the corresponding data is manipulated and tagged appropriately.

The following description is intended to illustrate how the present invention could be used to compartmentalize a setup comprising an externally facing Apache Web-server configured to delegate the handling of Java servlets or the serving of JSP files to two separate instances Jakarta/Tomcat, each running in its own compartment. By default, each compartment uses a *chroot*-ed filesystem so as not to interfere with the other compartments.

Figure 10 of the drawings illustrates schematically the Apache processes residing in one compartment (WEB). This compartment is externally accessible using the rule:

```
HOST* -> COMPARTMENT WEB
        METHOD TCP PORT 80 NETDEV eth0
```

The presence of the NETDEV component in the rule specifies the network-interfaces which Apache is allowed to use. This is useful for restricting Apache to using only the external interface on dual/multi-homed gateway systems. This is intended to prevent a compromised instance of Apache being used to launch attacks on back-end networks through internally facing network interfaces. The WEB compartment is allowed to communicate to two separate instances of Jakarta/Tomcat (TOMCAT1 and TOMCAT2) via two rules which take the form:

```
COMPARTMENT:WEB -> COMPARTMENT:TOMCAT1
```

METHOD TCP PORT 8007

COMPARTMENT:WEB -> COMPARTMENT TOMCAT2
METHOD TCP PORT 8008

The servlets in TOMCAT1 are allowed to access a back-end host called Server1 using this rule:

COMPARTMENT:TOMCAT1 -> HOST:SERVER1
METHOD TCP .....

However, TOMCAT 2 is not allowed to access any back-end hosts at all - which is reflected by the absence of any additional rules. The kernel will deny any such attempt from TOMCAT2. This allows one to selectively alter the view of a back-end network depending on which services are being hosted, and to restrict the visibility of back-end hosts on a per-compartment basis.

It is worth noting that the above four rules are all that is needed for this exemplary configuration. In the absence of any other rules, the servlets executing in the Java VM cannot initiate outgoing connections; in particular, it cannot be used to launch attacks on the internal back-end network on interface eth1. In addition, it may not access resources from other compartments (e.g. shared-memory segments, UNIX-domain sockets, etc.), nor be reached directly by remote hosts. In this case, mandatory restrictions have been placed on the behavior of Apache and Jakarta/Tomcat without recompiling or modifying their sources.

An example of application integration will now be described with reference to OpenMail 6.0. The OpenMail 6.0 distribution for Linux consists of a large 160Mb+ archive of some unspecified format, and an install-script *ominstall*. To install OpenMail, it is first necessary to chroot to an allocated bare-bones inner-compartment:

```
root@tlinux#    chroot /compt/omailin
root@tlinux#    ominstall
```

root@tlinux# [Wait for OpenMail to install naturally]
root@tlinux# [Do additional configuration if required, e.g. set up mailnodes]

Since OpenMail 6.0 has a Web-based interface which is also required to be installed, another bare-bones compartment is allocated (omailout) and an Apache HTTP-server is installed o handle the HTTP queries:

root@tlinux# chroot /compt/omailout
root@tlinux# rpm --install <apache-RPM-filoename>
root@tlinux# Configure Apache's httpd.conf to handle CGI-requests as required by OpenMail's installation instructions]

At this point, it is also necessary to install the CGI-binaries which come with OpenMail 6.0 so that they can be accessed by the Apache HTTP-server. This can be done by one of two methods:

* Install OpenMail again in *omailout* and remove unnecessary portions, e.g. server-processes; or
* Copy the OpenMail CGI-binaries from *omailin*, taking care to preserve permissions and directory structure.

In either case, the CGI-binaries typically are placed in the cgi-bin directory of the Apache Web-server. If disk-space is not an issue, the former approach is more brute-force and works well. The latter method can be used if it is necessary to be sure of exactly which binaries are to be placed in the externally-facing *omailout* compartment. Finally, both compartments can be started:

root@tlinux# comp_start omailout omailin

It may be possible that IP fragments are received with different originating compartment numbers. In such a case, the system may include means for disallowing fragment re-assembly to proceed with fragments of differing compartment numbers.

Support for various other network protocols may be included, e.g. IPX/SPX, etc.

It is envisaged that a more comprehensive method for filesystem protection than chroot-jails might be used.

Referring to Figure 13 of the drawings, the operation of an exemplary embodiment of the present invention is illustrated schematically. A gateway system 600 (connected to both an internal and external network) is shown. The gateway system 600 is hosting multiple types of services Service0, Service1,....., ServiceN, each of which is connected to some specified back-end host, Host0, Host1,.....HostX, HostN, to perform its function, e.g. retrieve records from a back-end database. Many back-end hosts may be present on an internal network at any one time (not all of which are intended to be accessible by the same set of services). It is essential that, if these server-processes are compromised, they should not be able to be used to probe other back-end hosts not originally intended to be used by the services. The present invention is intended to limit the damage an attacker can do by restricting the visibility of hosts on the same network.

In Figure 13, Service0 and Service1 are only allowed to access the network Subnet1 through the network-interface eth0. Therefore, attempts to access Host0/Host1 succeed because they are Subnet1, but attempts to access Subnet2 via eth1 fail. Further, ServiceN is allowed to access only HostX on eth1. Thus any attempt by ServiceN to access HostN fails, even if HostN is on the same subnet as HostX, and any attempt by ServiceN to access any host on Subnet1 fails.

The restrictions can be specified (by rules or routing-tables) by subnet or by specific host, which in turn may also be qualified by a specific subnet.

Referring to Figure 14 of the drawings, the operation of an operating system according to an exemplary embodiment of the fourth aspect of the invention of our first co-pending International Application is illustrated schematically. The main preferred features of an exemplary embodiment of this aspect of the invention are:

1. Modifications to the source code of the operating system in the areas in which transitions to root are possible. Hooks are added to these points so that, at run-time, these call out to functions that either allow or deny the transition to take place.

2. Modifications to the source code of the operating system to mark each running process with a tag. As described above, processes which are spawned inherit their tag from their parent process. Special privileged programs can launch an external program with a tag different from its own (the means by which the system is populated with processes with different tags).

3. A mechanism by which a configuration-utility can specify to the operating system at run-time which processes associated with a particular tag are to be marked as "sealed".

4. Configuration files describing data to be passed to the configuration-utility described above.

The present invention thus provides a trusted operating system, particularly Linux-based, in which the functionality is largely provided at the kernel level with a path-based specification of rules which are not accessed when files or programs are accessed. This is achieved by inferring any administrative privilege on running processes rather than on programs or files stored on disk. Such privileges are conferred by the inheritance of an administrative tag or label upon activation and thus there is no need to subsequently decode streams or packets for embedded security attributes, since streams or packets are not re-routed along different paths according to their security attributes.

Linux functionality is accessible without the need for trusted applications in user space and there is no requirement to upgrade or downgrade or otherwise modify security levels on running programs.

Embodiments of the present invention have been described above by way of examples only and it will be apparent to a person skilled in the art that modifications and variations can be made to the described embodiments without departing from the scope of the invention as defined by the appended claims.

Claims

1) A gateway system having a dual interface connected to both internal and external networks for hosting a plurality of services running processes, the system providing at least some of said running processes with a tag or label indicative of a logically protected computing compartment, said processes having the same tag or label belonging to the same compartment, the system further defining specific communications paths between said compartments and other networks, and permitting communication between a compartment and a host or a said network only in the event that a communication path or interface connection therebetween is defined.

2) A gateway system according to claim 1, in which the networks are local networks.

3) A gateway system according to claims 1 or 2, in which the networks are remote networks.

4) A gateway system according to claim 1, in which the communication path comprises an interface connection.

5) A gateway system according to claim 1, including access control checks which consult rules specifying which classes of processors are allowed to access which networks or hosts, in order to determine whether or not a communication path or interface connection therebetween is defined.

6) A gateway system according to claim 5, in which the access control checks are performed in an operating system of the gateway system.

7) A gateway system according to claim 6, in which the access control checks are performed in a kernel of the operating system of the gateway system.

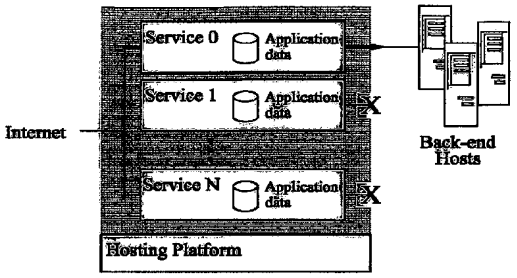
8) A gateway system according to claim 5, comprising a kernel which is provided with means for attaching a tag or label for each running process, the tag or label indicating

notionally which compartment or process belongs to.

9) A gateway system according to claim 8, comprising a system administrator which specifies rules defining the permitted communication paths between compartments and said hosts or networks.

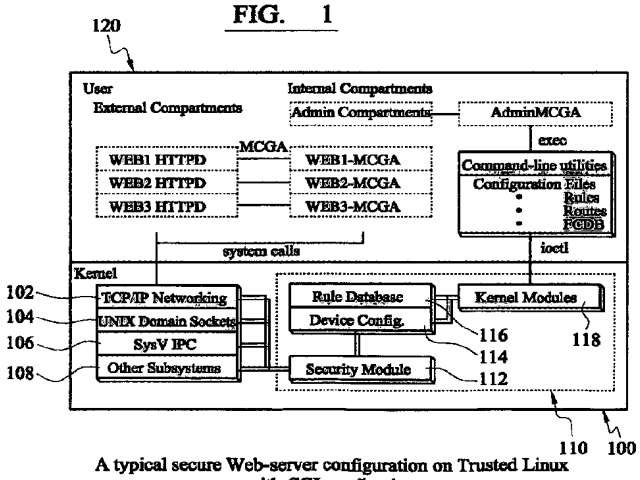
10) A gateway system according to claim 8, wherein a separate rooting-table per-compartment is provided which defines the permitted communication paths between compartments and said hosts or networks.

10) A gateway system according to claim 8, in which said running processes are a thread.



Example architecture for multi-service hosting on an operating system with the containment property.

FIG. 1



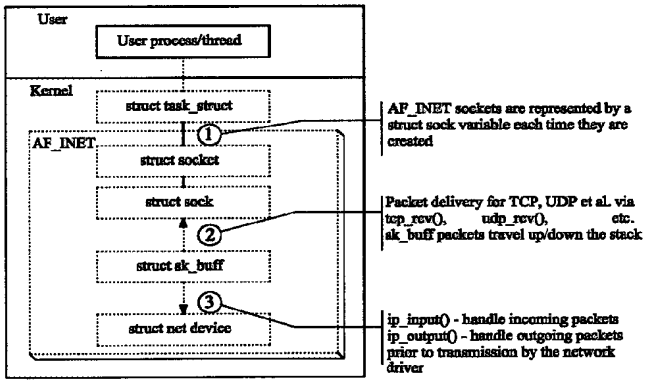
A typical secure Web-server configuration on Trusted Linux with CGI-sandboxing

FIG. 2

```
struct csec info {
        unsigned long sl ;
} ;
struct sock {
...
#ifdef CASPER
        struct csecinfo csi : /* contains compartment number */
#endif /* CASPER */
} ;
```

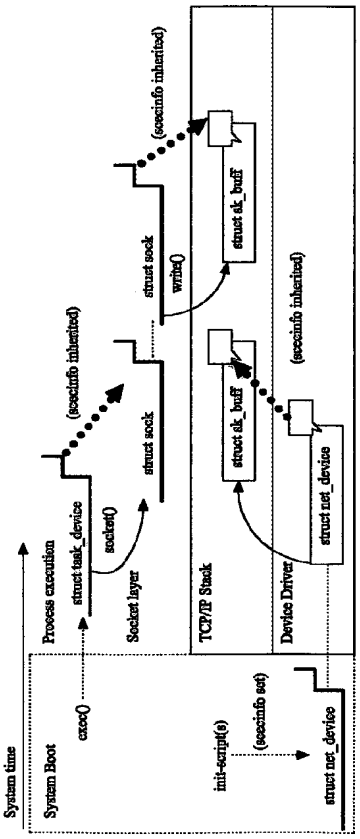
Example of modified datatype

FIG. 3



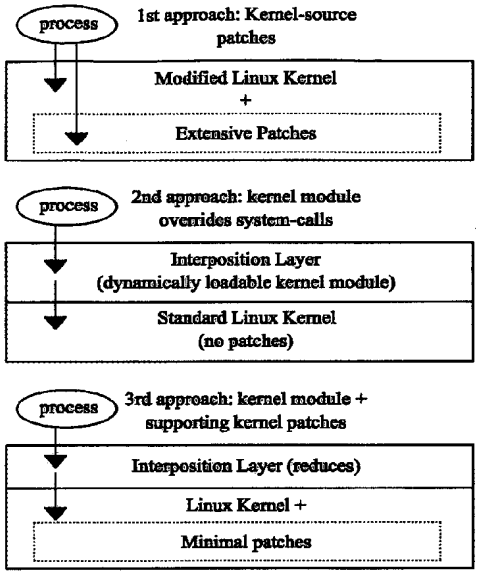
Major networking datatypes in Linux IP networking

FIG. 4



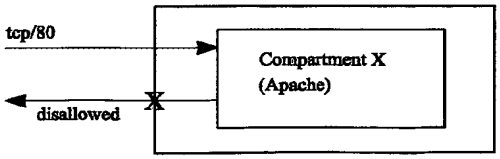
Propagation of struct secinfo data-members for IP-networking

FIG. 5



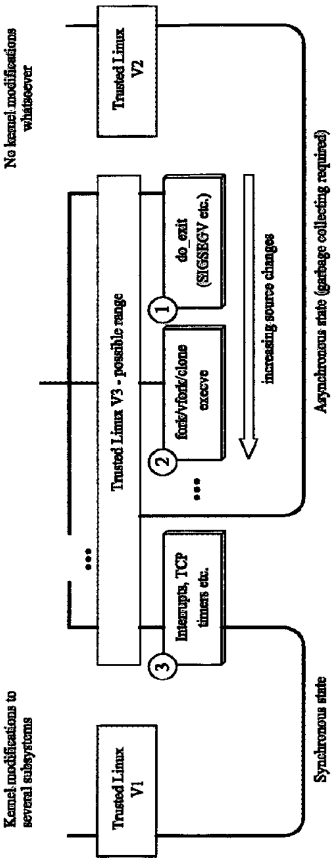
Three approaches to building containment into the Linux kernel

FIG. 6



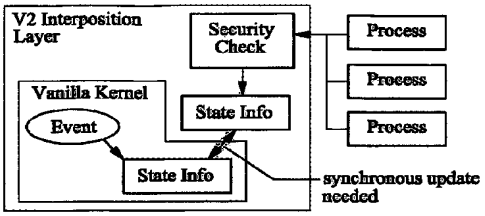
Only incoming TCP connections allowed

FIG. 7



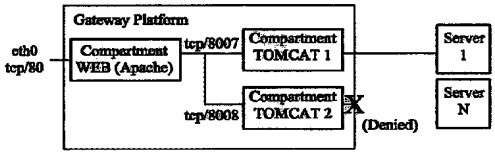
Spectrum of options available for the construction of a hybrid containment prototype OS

FIG. 8



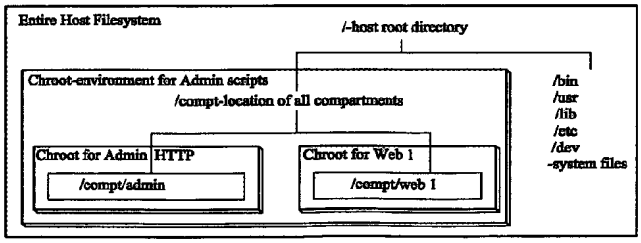
The need to update replicated kernel state in synchrony

FIG. 9



Configuration of Apache and Tomcat Java VMs

FIG. 10



Layered chroot-ed environments in Trusted Linux

FIG. 11

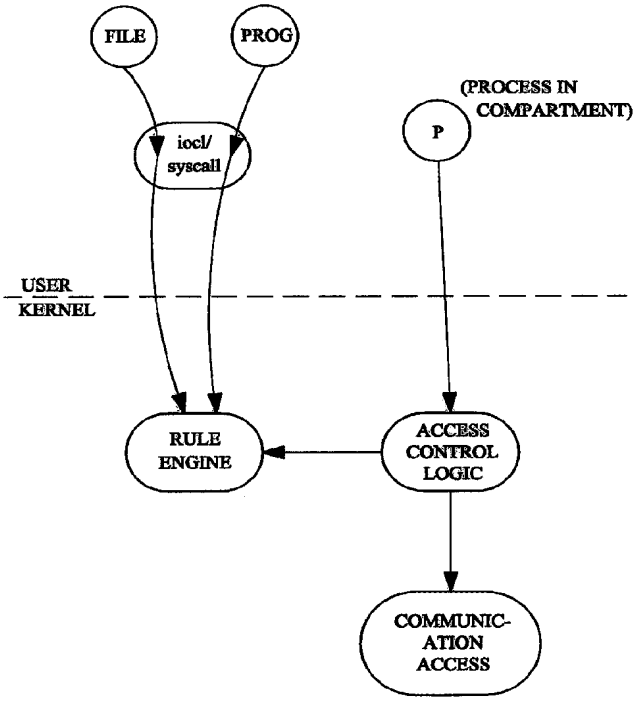

FILE
PROG
(PROCESS IN COMPARTMENT)
P
iocl/ syscall
USER
KERNEL
RULE ENGINE
ACCESS CONTROL LOGIC
COMMUNIC-ATION ACCESS

FIG. 12

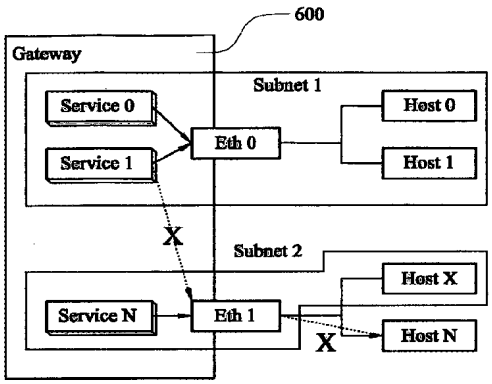

600
Gateway
Subnet 1
Service 0
Service 1
Eth 0
Host 0
Host 1
X
Subnet 2
Host X
Service N
Eth 1
X
Host N

FIG. 13

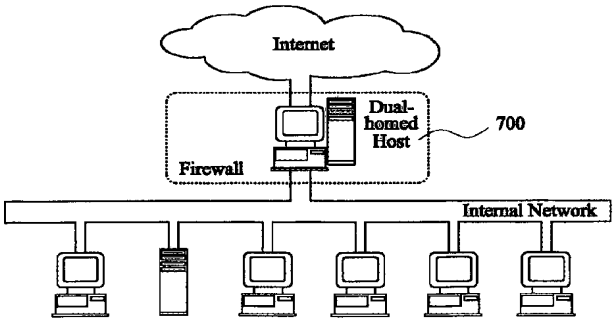

Internet
Dual-
homed
Host
700
Firewall
Internal Network

FIG. 15

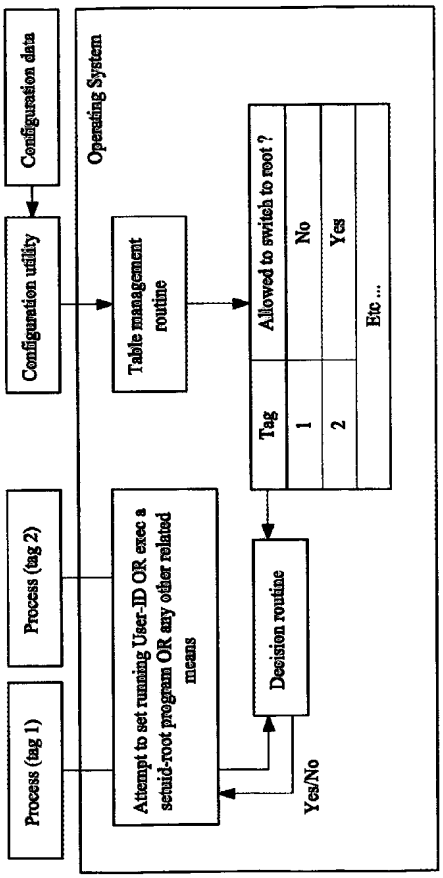

Configuration data
Configuration utility
Operating System
Table management routine
Allowed to switch to root ?
Tag
1
2
No
Yes
Etc ...
Process (tag 2)
Process (tag 1)
Attempt to set running User-ID OR exec a setuid-root program OR any other related means
Decision routine
Yes/No

FIG. 14

【国際調査報告】

INTERNATIONAL SEARCH REPORT

PCT/GB 02/00385

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F1/00

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)
EPO-Internal, WPI Data, PAJ

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	ANONYMOUS: "Secure Execution Environments, Internet Safety Through Type-Enforcing Firewalls" INTERNET ARTICLE, 'Online! 15 August 2000 (2000-08-15), XP002197021 Retrieved from the Internet: <URL:http://www.pgp.com/research/nailabs/secure-execution/internet-safety.asp> 'retrieved on 2002-04-24! abstract page 1 -page 2; figures 1,2 -/--	1-11

[X] Further documents are listed in the continuation of box C. [ ] Patent family members are listed in annex.

* Special categories of cited documents :
A document defining the general state of the art which is not considered to be of particular relevance
E earlier document but published on or after the international filing date
L document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
O document referring to an oral disclosure, use, exhibition or other means
P document published prior to the international filing date but later than the priority date claimed
T later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
X document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
Y document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.
& document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
24 April 2002	13/05/2002

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Kerschbaumer, J

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

PCT/GB 02/00385

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication,where appropriate, of the relevant passages	Relevant to claim No.
X	SERGE E. HALLYN, PHIL KEARNS: "Domain and Type Enforcement for Linux" INTERNET ARTICLE, 'Online! 14 October 2000 (2000-10-14), XP002197019 This paper was originally published in the Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta October 10-14, 2000 Retrieved from the Internet: <URL:http://www.usenix.org/publications/library/proceedings/als2000/full_papers/hallyn/hallyn_html/> 'retrieved on 2002-04-24! abstract page 1, paragraph 2 - paragraph 3 page 2, paragraph 2 - paragraph 3 page 3, paragraph 3 page 5, paragraph 1 page 8, paragraph 5 ---	1-11
X	PETER LOSCOCCO, STEPHEN SMALLEY: "Integrating Flexible Support for Security Policies into the Linux Operating System" INTERNET ARTICLE, 'Online! 2 January 2001 (2001-01-02), XP002197020 Retrieved from the Internet: <URL:www.nsa.gov/selinux (mirror: http://the.wiretapped.net/security/operating-systems/selinux/papers/slinux-200101020953.pdf)> 'retrieved on 2002-04-24! abstract page 3 -page 5 ---	1-11
A	DANIEL SENIE: "Using the SOCK_PACKET mechanism in Linux To Gain Complete Control of an Ethernet Interface" INTERNET ARTICLE, 'Online! 18 February 1999 (1999-02-18), XP002197022 Retrieved from the Internet: <URL:http://www.senie.com/dan/technology/sock_packet.html> 'retrieved on 2002-04-24! abstract page 2, paragraph 3 -----	1-11

Form PCT/ISA/210 (continuation of second sheet) (July 1992)

フロントページの続き

(72)発明者 チョー、ツェ・フオン
イギリス国 ビーエス３２ ８エービー ブリストル、ブラッドリー・ストーク、ザ・カルヴァート ４６
(72)発明者 ダルトン、クリストファー・アイ
イギリス国 ビーエス６ ６ティジェイ ブリストル、レッドランド、バーリントン・ロード １９
(72)発明者 ノーマン、アンドリュー・パトリック
イギリス国 ビーエス３２ ０ディアール ブリストル、ブラッドリー・ストーク、ジュニパー・ウェイ ２５４

Ｆターム(参考) 5B085 AA08 AE08 AE23 BG01 BG02 BG07
5B089 KA17 KG05 KG08